The Python/C API

Wydanie 3.13.0

Guido van Rossum and the Python development team

października 17, 2024

Python Software Foundation
Email: docs@python.org

Spis treści

1 Wprowadzenie **3**
1.1 Coding standards . . . 3
1.2 Pliki Włączania - z ang. Include . . . 3
1.3 Useful macros . . . 4
1.4 Przedmioty, ich Rodzaje i Liczby Odwołań . . . 7
1.4.1 Liczby odniesień . . . 7
1.4.2 Typy . . . 10
1.5 Wyjątki . . . 10
1.6 Załączanie programu interpretującego język pytonowski . . . 12
1.7 Odpluskwiające Budowy . . . 13

2 C API Stability **15**
2.1 Unstable C API . . . 15
2.2 Stable Application Binary Interface . . . 15
2.2.1 Limited C API . . . 16
2.2.2 stabilnego ABI . . . 16
2.2.3 Limited API Scope and Performance . . . 16
2.2.4 Limited API Caveats . . . 16
2.3 Platform Considerations . . . 17
2.4 Contents of Limited API . . . 17

3 The Very High Level Layer **43**

4 Reference Counting **47**

5 Obsługa sytuacji wyjątkowych **51**
5.1 Printing and clearing . . . 51
5.2 Raising exceptions . . . 52
5.3 Issuing warnings . . . 55
5.4 Querying the error indicator . . . 55
5.5 Signal Handling . . . 59
5.6 Exception Classes . . . 60
5.7 Przedmioty Sytuacji Wyjątkowych . . . 60
5.8 Unicode Exception Objects . . . 61
5.9 Kontrola Rekursji . . . 62
5.10 Sztandarowe Sytuacje Wyjątkowe . . . 63
5.11 Standard Warning Categories . . . 64

6 Utilities **65**
6.1 Operating System Utilities . . . 65
6.2 System Functions . . . 68

6.3 Process Control . . . 70
6.4 Importing Modules . . . 70
6.5 Data marshalling support . . . 74
6.6 Pobieranie kolejnych rzeczy podanych na wejściu i konstruowanie wartości. . . . 75
6.6.1 Parsowanie argumentów . . . 75
6.6.2 Budowanie wartości . . . 81
6.7 String conversion and formatting . . . 83
6.8 PyHash API . . . 85
6.9 Reflection . . . 86
6.10 Codec registry and support functions . . . 87
6.10.1 Codec lookup API . . . 88
6.10.2 Registry API for Unicode encoding error handlers . . . 88
6.11 PyTime C API . . . 89
6.11.1 Typy . . . 89
6.11.2 Clock Functions . . . 89
6.11.3 Raw Clock Functions . . . 90
6.11.4 Conversion functions . . . 90
6.12 Support for Perf Maps . . . 90

7 Warstwa obiektów abstrakcyjnych 93
7.1 Object Protocol . . . 93
7.2 Call Protocol . . . 100
7.2.1 The *tp_call* Protocol . . . 100
7.2.2 The Vectorcall Protocol . . . 100
7.2.3 Object Calling API . . . 102
7.2.4 Call Support API . . . 104
7.3 Number Protocol . . . 104
7.4 Sequence Protocol . . . 108
7.5 Mapping Protocol . . . 109
7.6 Iterator Protocol . . . 111
7.7 Buffer Protocol . . . 112
7.7.1 Buffer structure . . . 113
7.7.2 Buffer request types . . . 114
7.7.3 Complex arrays . . . 116
7.7.4 Buffer-related functions . . . 117

8 Concrete Objects Layer 119
8.1 Fundamental Objects . . . 119
8.1.1 Type Objects . . . 119
8.1.2 The `None` Object . . . 125
8.2 Numeric Objects . . . 125
8.2.1 Integer Objects . . . 125
8.2.2 Obiekty logiczne . . . 132
8.2.3 Floating-Point Objects . . . 133
8.2.4 Obiekt Liczby Zespolonej . . . 134
8.3 Sequence Objects . . . 136
8.3.1 Bytes Objects . . . 136
8.3.2 Byte Array Objects . . . 138
8.3.3 Unicode Objects and Codecs . . . 139
8.3.4 Tuple Objects . . . 156
8.3.5 Struct Sequence Objects . . . 158
8.3.6 List Objects . . . 159
8.4 Container Objects . . . 161
8.4.1 Obiekty słownika . . . 161
8.4.2 Set Objects . . . 166
8.5 Obiekty Funkcja . . . 168
8.5.1 Obiekty Funkcja . . . 168
8.5.2 Instance Method Objects . . . 170

8.5.3 Method Objects . . . 170
8.5.4 Cell Objects . . . 170
8.5.5 Code Objects . . . 171
8.5.6 Extra information . . . 174
8.6 Other Objects . . . 175
8.6.1 File Objects . . . 175
8.6.2 Module Objects . . . 176
8.6.3 Iterator Objects . . . 184
8.6.4 Descriptor Objects . . . 185
8.6.5 Slice Objects . . . 185
8.6.6 MemoryView objects . . . 187
8.6.7 Weak Reference Objects . . . 187
8.6.8 Capsules . . . 189
8.6.9 Frame Objects . . . 190
8.6.10 Generator Objects . . . 193
8.6.11 Coroutine Objects . . . 193
8.6.12 Context Variables Objects . . . 193
8.6.13 DateTime Objects . . . 195
8.6.14 Objects for Type Hinting . . . 198

9 Initialization, Finalization, and Threads 201
9.1 Before Python Initialization . . . 201
9.2 Global configuration variables . . . 202
9.3 Initializing and finalizing the interpreter . . . 205
9.4 Process-wide parameters . . . 207
9.5 Thread State and the Global Interpreter Lock . . . 211
9.5.1 Releasing the GIL from extension code . . . 211
9.5.2 Non-Python created threads . . . 212
9.5.3 Cautions about fork() . . . 212
9.5.4 High-level API . . . 213
9.5.5 Low-level API . . . 215
9.6 Sub-interpreter support . . . 218
9.6.1 A Per-Interpreter GIL . . . 220
9.6.2 Bugs and caveats . . . 220
9.7 Asynchronous Notifications . . . 221
9.8 Profiling and Tracing . . . 221
9.9 Reference tracing . . . 223
9.10 Advanced Debugger Support . . . 224
9.11 Thread Local Storage Support . . . 224
9.11.1 Thread Specific Storage (TSS) API . . . 224
9.11.2 Thread Local Storage (TLS) API . . . 225
9.12 Synchronization Primitives . . . 226
9.12.1 Python Critical Section API . . . 227

10 Python Initialization Configuration 229
10.1 Example . . . 229
10.2 PyWideStringList . . . 230
10.3 PyStatus . . . 230
10.4 PyPreConfig . . . 232
10.5 Preinitialize Python with PyPreConfig . . . 233
10.6 PyConfig . . . 234
10.7 Initialization with PyConfig . . . 245
10.8 Isolated Configuration . . . 247
10.9 Python Configuration . . . 247
10.10 Python Path Configuration . . . 247
10.11 Py_GetArgcArgv() . . . 249
10.12 Multi-Phase Initialization Private Provisional API . . . 249

11 Zarządzanie Pamięcią **251**
11.1 Skorowidz . . . 251
11.2 Allocator Domains . . . 252
11.3 Raw Memory Interface . . . 252
11.4 Sprzęg Pamięci . . . 253
11.5 Object allocators . . . 255
11.6 Default Memory Allocators . . . 255
11.7 Customize Memory Allocators . . . 256
11.8 Debug hooks on the Python memory allocators . . . 257
11.9 The pymalloc allocator . . . 259
11.9.1 Customize pymalloc Arena Allocator . . . 259
11.10 The mimalloc allocator . . . 259
11.11 tracemalloc C API . . . 260
11.12 Przykłady . . . 260

12 Object Implementation Support **263**
12.1 Przydzielanie obiektów na stercie . . . 263
12.2 Wspólne struktury obiektów . . . 264
12.2.1 Base object types and macros . . . 264
12.2.2 Implementing functions and methods . . . 266
12.2.3 Accessing attributes of extension types . . . 269
12.3 Type Objects . . . 272
12.3.1 Quick Reference . . . 273
12.3.2 PyTypeObject Definition . . . 277
12.3.3 PyObject Slots . . . 278
12.3.4 PyVarObject Slots . . . 279
12.3.5 PyTypeObject Slots . . . 279
12.3.6 Static Types . . . 298
12.3.7 Heap Types . . . 298
12.4 Number Object Structures . . . 298
12.5 Mapping Object Structures . . . 301
12.6 Sequence Object Structures . . . 301
12.7 Buffer Object Structures . . . 302
12.8 Async Object Structures . . . 303
12.9 Slot Type typedefs . . . 304
12.10 Przykłady . . . 305
12.11 Supporting Cyclic Garbage Collection . . . 308
12.11.1 Controlling the Garbage Collector State . . . 310
12.11.2 Querying Garbage Collector State . . . 310

13 API i wersjonowanie ABI **313**

14 Monitoring C API **315**

15 Generating Execution Events **317**
15.1 Managing the Monitoring State . . . 318

A Słownik **321**

B O tej dokumentacji **337**
B.1 Współtwórcy dokumentacji Pythona . . . 337

C Historia i zapisy prawne **339**
C.1 Historia programu . . . 339
C.2 Zasady i warunki postępowania z Pythonem i ogólnie jego użycia . . . 340
C.2.1 PSF LICENSE AGREEMENT FOR PYTHON 3.13.0 . . . 340
C.2.2 BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0 . . . 341
C.2.3 CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1 . . . 342
C.2.4 CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2 . . . 343

C.2.5 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON 3.13.0 DOCUMENTATION . . . 343
C.3 Licenses and Acknowledgements for Incorporated Software . . . 344
C.3.1 Mersenne Twister . . . 344
C.3.2 Sockets . . . 345
C.3.3 Asynchronous socket services . . . 345
C.3.4 Cookie management . . . 346
C.3.5 Execution tracing . . . 346
C.3.6 UUencode and UUdecode functions . . . 347
C.3.7 XML Remote Procedure Calls . . . 347
C.3.8 test_epoll . . . 348
C.3.9 Select kqueue . . . 348
C.3.10 SipHash24 . . . 349
C.3.11 strtod and dtoa . . . 349
C.3.12 OpenSSL . . . 350
C.3.13 expat . . . 353
C.3.14 libffi . . . 353
C.3.15 zlib . . . 354
C.3.16 cfuhash . . . 354
C.3.17 libmpdec . . . 355
C.3.18 W3C C14N test suite . . . 356
C.3.19 mimalloc . . . 356
C.3.20 asyncio . . . 357
C.3.21 Global Unbounded Sequences (GUS) . . . 357

D Prawa autorskie **359**

Indeks **361**

This manual documents the API used by C and C++ programmers who want to write extension modules or embed Python. It is a companion to extending-index, which describes the general principles of extension writing but does not document the API functions in detail.

ROZDZIAŁ 1

Wprowadzenie

Interfejs programowania aplikacji w Pythonie daje programistom języków C i C++ dostęp do programu interpretującego polecenia języka pytonowskiego na wielu poziomach. Sprzęg (API) jest równo użyteczny z poziomu C++ ale dla porządku jest zwykle określany mianem sprzęgu pomiędzy językami pytonowskim a C (z ang. - Python/C API). Istnieją dwie zasadniczo różne przyczyny dla użycia interfejsu między językami Python i C. Pierwszą przyczyną jest pisanie *modułów rozszerzających* dla szczególnych powodów; są to moduły języka C, które rozszerzają interpreter Pythona. To jest zwykle najczęstsze użycie. Drugą przyczyną jest użycie Pythona jako komponentu większego programu; ta technika jest zwykle określana mianem załączania - z ang. - *embedding* w aplikacji.

Writing an extension module is a relatively well-understood process, where a „cookbook" approach works well. There are several tools that automate the process to some extent. While people have embedded Python in other applications since its early existence, the process of embedding Python is less straightforward than writing an extension.

Wiele zadań sprzęgu (API) jest użytecznych niezależnie od tego czy załączasz, czy też rozszerzasz program interpretujący język pytonowski; co więcej, większość aplikacji które załącza program interpretujący polecenia jezyka pytonowskiego potrzebuje także szczególnych rozszerzeń, więc prawdopodobnie jest dobrym pomysłem zaznajomienie się z pisaniem rozszerzenia przed próbą załączenia języka pytonowskiego w prawdziwej aplikacji.

1.1 Coding standards

If you're writing C code for inclusion in CPython, you **must** follow the guidelines and standards defined in **PEP 7**. These guidelines apply regardless of the version of Python you are contributing to. Following these conventions is not necessary for your own third party extension modules, unless you eventually expect to contribute them to Python.

1.2 Pliki Włączania - z ang. Include

Wszystkie zadania, definicje typu i makropoleceń konieczne do użycia sprzęgu między językami pytonowskim i C są włączane do źródeł w kodzie użytkownika przez następującą linijkę:

```
#define PY_SSIZE_T_CLEAN
#include <Python.h>
```

This implies inclusion of the following standard headers: `<stdio.h>`, `<string.h>`, `<errno.h>`, `<limits.h>`, `<assert.h>` and `<stdlib.h>` (if available).

Informacja

Jako że Python może definiować pewne definicje preprocesora, które wpływają na pliki nagłówkowe na niektórych systemach, *musisz* załączyć plik `Python.h` przed jakimikolwiek standardowymi nagłówkami.

It is recommended to always define `PY_SSIZE_T_CLEAN` before including `Python.h`. See *Pobieranie kolejnych rzeczy podanych na wejściu i konstruowanie wartości.* for a description of this macro.

Wszystkie widoczne dla użytkownika nazwy określone w Python.h (z wyjątkiem tych określonych przez załączone standardowe pliki nagłówkowe) mają jeden z przedrostków `Py` lub `_Py`. Nazwy rozpoczynające się od `_Py` służą do wewnętrznego użytku przez urzeczywistnienie programu interpretującego języka pytonowskiego i nie powinno być używane przez piszących rozszerzenia. Nazwy członków struktury nie mają zarezerwowanych przedrostków.

Informacja

User code should never define names that begin with `Py` or `_Py`. This confuses the reader, and jeopardizes the portability of the user code to future Python versions, which may define additional names beginning with one of these prefixes.

The header files are typically installed with Python. On Unix, these are located in the directories *prefix*`/include/pythonversion/` and *exec_prefix*`/include/pythonversion/`, where `prefix` and `exec_prefix` are defined by the corresponding parameters to Python's **`configure`** script and *version* is `'%d.%d' % sys.version_info[:2]`. On Windows, the headers are installed in *prefix*`/include`, where `prefix` is the installation directory specified to the installer.

To include the headers, place both directories (if different) on your compiler's search path for includes. Do *not* place the parent directories on the search path and then use `#include <pythonX.Y/Python.h>`; this will break on multi-platform builds since the platform independent headers under `prefix` include the platform specific headers from `exec_prefix`.

C++ users should note that although the API is defined entirely using C, the header files properly declare the entry points to be `extern "C"`. As a result, there is no need to do anything special to use the API from C++.

1.3 Useful macros

Several useful macros are defined in the Python header files. Many are defined closer to where they are useful (e.g. *`Py_RETURN_NONE`*). Others of a more general utility are defined here. This is not necessarily a complete listing.

`PyMODINIT_FUNC`

Declare an extension module `PyInit` initialization function. The function return type is *`PyObject`*`*`. The macro declares any special linkage declarations required by the platform, and for C++ declares the function as `extern "C"`.

The initialization function must be named `PyInit_`*name*, where *name* is the name of the module, and should be the only non-`static` item defined in the module file. Example:

```
static struct PyModuleDef spam_module = {
    PyModuleDef_HEAD_INIT,
    .m_name = "spam",
    ...
};

PyMODINIT_FUNC
PyInit_spam(void)
{
    return PyModule_Create(&spam_module);
}
```

Py_ABS(x)

Return the absolute value of `x`.

Added in version 3.3.

Py_ALWAYS_INLINE

Ask the compiler to always inline a static inline function. The compiler can ignore it and decides to not inline the function.

It can be used to inline performance critical static inline functions when building Python in debug mode with function inlining disabled. For example, MSC disables function inlining when building in debug mode.

Marking blindly a static inline function with Py_ALWAYS_INLINE can result in worse performances (due to increased code size for example). The compiler is usually smarter than the developer for the cost/benefit analysis.

If Python is built in debug mode (if the *Py_DEBUG* macro is defined), the *Py_ALWAYS_INLINE* macro does nothing.

It must be specified before the function return type. Usage:

```
static inline Py_ALWAYS_INLINE int random(void) { return 4; }
```

Added in version 3.11.

Py_CHARMASK(c)

Argument must be a character or an integer in the range [-128, 127] or [0, 255]. This macro returns `c` cast to an `unsigned char`.

Py_DEPRECATED(version)

Use this for deprecated declarations. The macro must be placed before the symbol name.

Example:

```
Py_DEPRECATED(3.8) PyAPI_FUNC(int) Py_OldFunction(void);
```

Zmienione w wersji 3.8: MSVC support was added.

Py_GETENV(s)

Like `getenv(s)`, but returns `NULL` if `-E` was passed on the command line (see *PyConfig.use_environment*).

Py_MAX(x, y)

Return the maximum value between `x` and `y`.

Added in version 3.3.

Py_MEMBER_SIZE(type, member)

Return the size of a structure (`type`) `member` in bytes.

Added in version 3.6.

Py_MIN(x, y)

Return the minimum value between `x` and `y`.

Added in version 3.3.

Py_NO_INLINE

Disable inlining on a function. For example, it reduces the C stack consumption: useful on LTO+PGO builds which heavily inline code (see bpo-33720).

Usage:

```
Py_NO_INLINE static int random(void) { return 4; }
```

Added in version 3.11.

Py_STRINGIFY(x)

Convert `x` to a C string. E.g. `Py_STRINGIFY(123)` returns `"123"`.

Added in version 3.4.

Py_UNREACHABLE()

Use this when you have a code path that cannot be reached by design. For example, in the `default:` clause in a `switch` statement for which all possible values are covered in `case` statements. Use this in places where you might be tempted to put an `assert(0)` or `abort()` call.

In release mode, the macro helps the compiler to optimize the code, and avoids a warning about unreachable code. For example, the macro is implemented with `__builtin_unreachable()` on GCC in release mode.

A use for `Py_UNREACHABLE()` is following a call a function that never returns but that is not declared `_Py_NO_RETURN`.

If a code path is very unlikely code but can be reached under exceptional case, this macro must not be used. For example, under low memory condition or if a system call returns a value out of the expected range. In this case, it's better to report the error to the caller. If the error cannot be reported to caller, *Py_FatalError()* can be used.

Added in version 3.7.

Py_UNUSED(arg)

Use this for unused arguments in a function definition to silence compiler warnings. Example: `int func(int a, int Py_UNUSED(b)) { return a; }`.

Added in version 3.4.

PyDoc_STRVAR(name, str)

Creates a variable with name `name` that can be used in docstrings. If Python is built without docstrings, the value will be empty.

Use *PyDoc_STRVAR* for docstrings to support building Python without docstrings, as specified in **PEP 7**.

Example:

```
PyDoc_STRVAR(pop_doc, "Remove and return the rightmost element.");

static PyMethodDef deque_methods[] = {
    // ...
    {"pop", (PyCFunction)deque_pop, METH_NOARGS, pop_doc},
    // ...
}
```

PyDoc_STR(str)

Creates a docstring for the given input string or an empty string if docstrings are disabled.

Use *PyDoc_STR* in specifying docstrings to support building Python without docstrings, as specified in **PEP 7**.

Example:

```
static PyMethodDef pysqlite_row_methods[] = {
    {"keys", (PyCFunction)pysqlite_row_keys, METH_NOARGS,
        PyDoc_STR("Returns the keys of the row.")},
    {NULL, NULL}
};
```

1.4 Przedmioty, ich Rodzaje i Liczby Odwołań

Most Python/C API functions have one or more arguments as well as a return value of type `PyObject*`. This type is a pointer to an opaque data type representing an arbitrary Python object. Since all Python object types are treated the same way by the Python language in most situations (e.g., assignments, scope rules, and argument passing), it is only fitting that they should be represented by a single C type. Almost all Python objects live on the heap: you never declare an automatic or static variable of type `PyObject`, only pointer variables of type `PyObject*` can be declared. The sole exception are the type objects; since these must never be deallocated, they are typically static `PyTypeObject` objects.

Wszystkie przedmioty języka pytonowskiego (nawet liczby całkowite języka pytonowskiego) mają *rodzaj* i *liczbę odniesień*. Typ przedmiotu określa jakiego rodzaju przedmiot to jest (np. liczba całkowita, lista, lub zadanie zdefiniowane przez użytkownika; jest wiele więcej jak wyjaśniono w types). Dla każdego z dobrze-znanych rodzajów istnieje makropolecenie sprawdzające czy przedmiot jest tego rodzaju; na przykład, `PyList_Check(a)` jest prawdziwe wtedy (i tylko wtedy) gdy przedmiot na który wskazuje *a* jest lista z języka pytonowskiego.

1.4.1 Liczby odniesień

The reference count is important because today's computers have a finite (and often severely limited) memory size; it counts how many different places there are that have a *strong reference* to an object. Such a place could be another object, or a global (or static) C variable, or a local variable in some C function. When the last *strong reference* to an object is released (i.e. its reference count becomes zero), the object is deallocated. If it contains references to other objects, those references are released. Those other objects may be deallocated in turn, if there are no more references to them, and so on. (There's an obvious problem with objects that reference each other here; for now, the solution is „don't do that.")

Reference counts are always manipulated explicitly. The normal way is to use the macro `Py_INCREF()` to take a new reference to an object (i.e. increment its reference count by one), and `Py_DECREF()` to release that reference (i.e. decrement the reference count by one). The `Py_DECREF()` macro is considerably more complex than the incref one, since it must check whether the reference count becomes zero and then cause the object's deallocator to be called. The deallocator is a function pointer contained in the object's type structure. The type-specific deallocator takes care of releasing references for other objects contained in the object if this is a compound object type, such as a list, as well as performing any additional finalization that's needed. There's no chance that the reference count can overflow; at least as many bits are used to hold the reference count as there are distinct memory locations in virtual memory (assuming `sizeof(Py_ssize_t) >= sizeof(void*)`). Thus, the reference count increment is a simple operation.

It is not necessary to hold a *strong reference* (i.e. increment the reference count) for every local variable that contains a pointer to an object. In theory, the object's reference count goes up by one when the variable is made to point to it and it goes down by one when the variable goes out of scope. However, these two cancel each other out, so at the end the reference count hasn't changed. The only real reason to use the reference count is to prevent the object from being deallocated as long as our variable is pointing to it. If we know that there is at least one other reference to the object that lives at least as long as our variable, there is no need to take a new *strong reference* (i.e. increment the reference count) temporarily. An important situation where this arises is in objects that are passed as arguments to C functions in an extension module that are called from Python; the call mechanism guarantees to hold a reference to every argument for the duration of the call.

However, a common pitfall is to extract an object from a list and hold on to it for a while without taking a new reference. Some other operation might conceivably remove the object from the list, releasing that reference, and possibly deallocating it. The real danger is that innocent-looking operations may invoke arbitrary Python code which could do this; there is a code path which allows control to flow back to the user from a `Py_DECREF()`, so almost any operation is potentially dangerous.

A safe approach is to always use the generic operations (functions whose name begins with `PyObject_`, `PyNumber_`, `PySequence_` or `PyMapping_`). These operations always create a new *strong reference* (i.e. increment the reference count) of the object they return. This leaves the caller with the responsibility to call `Py_DECREF()` when they are done with the result; this soon becomes second nature.

Szczegóły Liczby Odniesień

The reference count behavior of functions in the Python/C API is best explained in terms of *ownership of references*. Ownership pertains to references, never to objects (objects are not owned: they are always shared). „Owning a reference" means being responsible for calling Py_DECREF on it when the reference is no longer needed. Ownership can also be transferred, meaning that the code that receives ownership of the reference then becomes responsible for eventually releasing it by calling `Py_DECREF()` or `Py_XDECREF()` when it's no longer needed—or passing on this responsibility (usually to its caller). When a function passes ownership of a reference on to its caller, the caller is said to receive a *new* reference. When no ownership is transferred, the caller is said to *borrow* the reference. Nothing needs to be done for a *borrowed reference*.

Idąc dalej, gdy wywołujące zadanie przekazuje odniesienie do przedmiotu, istnieją dwie możliwości: zadanie *kradnie* odniesienie do przedmiotu, lub nie kradnie go. *Kradnięcie odniesienia* oznacza, że gdy przekazujesz odniesienie do zadania, to zadanie przyjmuje, że teraz ono posiada odniesienie i nie jesteś za nie odpowiedzialny ani chwili dłużej.

Few functions steal references; the two notable exceptions are `PyList_SetItem()` and `PyTuple_SetItem()`, which steal a reference to the item (but not to the tuple or list into which the item is put!). These functions were designed to steal a reference because of a common idiom for populating a tuple or list with newly created objects; for example, the code to create the tuple `(1, 2, "three")` could look like this (forgetting about error handling for the moment; a better way to code this is shown below):

```
PyObject *t;

t = PyTuple_New(3);
PyTuple_SetItem(t, 0, PyLong_FromLong(1L));
PyTuple_SetItem(t, 1, PyLong_FromLong(2L));
PyTuple_SetItem(t, 2, PyUnicode_FromString("three"));
```

Here, `PyLong_FromLong()` returns a new reference which is immediately stolen by `PyTuple_SetItem()`. When you want to keep using an object although the reference to it will be stolen, use `Py_INCREF()` to grab another reference before calling the reference-stealing function.

Incidentally, `PyTuple_SetItem()` is the *only* way to set tuple items; `PySequence_SetItem()` and `PyObject_SetItem()` refuse to do this since tuples are an immutable data type. You should only use `PyTuple_SetItem()` for tuples that you are creating yourself.

Equivalent code for populating a list can be written using `PyList_New()` and `PyList_SetItem()`.

However, in practice, you will rarely use these ways of creating and populating a tuple or list. There's a generic function, `Py_BuildValue()`, that can create most common objects from C values, directed by a *format string*. For example, the above two blocks of code could be replaced by the following (which also takes care of the error checking):

```
PyObject *tuple, *list;

tuple = Py_BuildValue("(iis)", 1, 2, "three");
list = Py_BuildValue("[iis]", 1, 2, "three");
```

It is much more common to use `PyObject_SetItem()` and friends with items whose references you are only borrowing, like arguments that were passed in to the function you are writing. In that case, their behaviour regarding references is much saner, since you don't have to take a new reference just so you can give that reference away („have it be stolen"). For example, this function sets all items of a list (actually, any mutable sequence) to a given item:

```
int
set_all(PyObject *target, PyObject *item)
{
    Py_ssize_t i, n;

    n = PyObject_Length(target);
    if (n < 0)
        return -1;
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
    for (i = 0; i < n; i++) {
        PyObject *index = PyLong_FromSsize_t(i);
        if (!index)
            return -1;
        if (PyObject_SetItem(target, index, item) < 0) {
            Py_DECREF(index);
            return -1;
        }
        Py_DECREF(index);
    }
    return 0;
}
```

The situation is slightly different for function return values. While passing a reference to most functions does not change your ownership responsibilities for that reference, many functions that return a reference to an object give you ownership of the reference. The reason is simple: in many cases, the returned object is created on the fly, and the reference you get is the only reference to the object. Therefore, the generic functions that return object references, like `PyObject_GetItem()` and `PySequence_GetItem()`, always return a new reference (the caller becomes the owner of the reference).

It is important to realize that whether you own a reference returned by a function depends on which function you call only — *the plumage* (the type of the object passed as an argument to the function) *doesn't enter into it!* Thus, if you extract an item from a list using `PyList_GetItem()`, you don't own the reference — but if you obtain the same item from the same list using `PySequence_GetItem()` (which happens to take exactly the same arguments), you do own a reference to the returned object.

Here is an example of how you could write a function that computes the sum of the items in a list of integers; once using `PyList_GetItem()`, and once using `PySequence_GetItem()`.

```
long
sum_list(PyObject *list)
{
    Py_ssize_t i, n;
    long total = 0, value;
    PyObject *item;

    n = PyList_Size(list);
    if (n < 0)
        return -1; /* Not a list */
    for (i = 0; i < n; i++) {
        item = PyList_GetItem(list, i); /* Can't fail */
        if (!PyLong_Check(item)) continue; /* Skip non-integers */
        value = PyLong_AsLong(item);
        if (value == -1 && PyErr_Occurred())
            /* Integer too big to fit in a C long, bail out */
            return -1;
        total += value;
    }
    return total;
}
```

```
long
sum_sequence(PyObject *sequence)
{
    Py_ssize_t i, n;
    long total = 0, value;
    PyObject *item;
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
    n = PySequence_Length(sequence);
    if (n < 0)
        return -1; /* Has no length */
    for (i = 0; i < n; i++) {
        item = PySequence_GetItem(sequence, i);
        if (item == NULL)
            return -1; /* Not a sequence, or other failure */
        if (PyLong_Check(item)) {
            value = PyLong_AsLong(item);
            Py_DECREF(item);
            if (value == -1 && PyErr_Occurred())
                /* Integer too big to fit in a C long, bail out */
                return -1;
            total += value;
        }
        else {
            Py_DECREF(item); /* Discard reference ownership */
        }
    }
    return total;
}
```

1.4.2 Typy

There are few other data types that play a significant role in the Python/C API; most are simple C types such as `int`, `long`, `double` and `char*`. A few structure types are used to describe static tables used to list the functions exported by a module or the data attributes of a new object type, and another is used to describe the value of a complex number. These will be discussed together with the functions that use them.

type **`Py_ssize_t`**

Część stabilnego ABI. A signed integral type such that `sizeof(Py_ssize_t) == sizeof(size_t)`. C99 doesn't define such a thing directly (size_t is an unsigned integral type). See **PEP 353** for details. `PY_SSIZE_T_MAX` is the largest positive value of type *`Py_ssize_t`*.

1.5 Wyjątki

Programujący komputer w języku pytonowskim musi sobie zaprzątać głowę tylko sytuacjami wyjątkowymi tylko jeśli szczególna obsługa błędów jest konieczna; Nieobsłużone wyjątki są automatycznie przesyłane do zadania wywołującego, potem do zadania które wywołało tamto zadanie, i tak dalej, dopóki nie natrafi na program interpretujący najwyższego poziomu, gdzie są przekazywane użytkownikowi wraz z wypisem kolejnych wywołań odłożonych na stercie.

For C programmers, however, error checking always has to be explicit. All functions in the Python/C API can raise exceptions, unless an explicit claim is made otherwise in a function's documentation. In general, when a function encounters an error, it sets an exception, discards any object references that it owns, and returns an error indicator. If not documented otherwise, this indicator is either `NULL` or `-1`, depending on the function's return type. A few functions return a Boolean true/false result, with false indicating an error. Very few functions return no explicit error indicator or have an ambiguous return value, and require explicit testing for errors with *`PyErr_Occurred()`*. These exceptions are always explicitly documented.

Exception state is maintained in per-thread storage (this is equivalent to using global storage in an unthreaded application). A thread can be in one of two states: an exception has occurred, or not. The function *`PyErr_Occurred()`* can be used to check for this: it returns a borrowed reference to the exception type object when an exception has occurred, and `NULL` otherwise. There are a number of functions to set the exception state: *`PyErr_SetString()`* is the most common (though not the most general) function to set the exception state, and *`PyErr_Clear()`* clears the exception state.

The full exception state consists of three objects (all of which can be `NULL`): the exception type, the corresponding exception value, and the traceback. These have the same meanings as the Python result of `sys.exc_info()`; however, they are not the same: the Python objects represent the last exception being handled by a Python `try` … `except` statement, while the C level exception state only exists while an exception is being passed on between C functions until it reaches the Python bytecode interpreter's main loop, which takes care of transferring it to `sys.exc_info()` and friends.

Zauważ że poczynając od języka pytonowskiego w wersji 1.5 preferowaną, bezpiecznym dla wątków sposobem na dostęp do stanu wyjątku z poziomu kodu napisanego w języku pytonowskim jest wezwanie zadania `sys.exc_info()`, które zwraca określony-dla-wątku stan wyjątku dla kodu napisanego w języku pytonowskim. Poza tym składnia obu sposobów na dostęp do stanu sytuacji wyjątkowej zmieniła się tak, że zadanie które złapie wyjątek zachowa i przywróci swój stan wyjątku tak, aby zachować stan wyjątku wywołujacego zadanie. To działanie zapobiega typowym błędom w obsłudze sytuacji wyjątkowych powodowanych przez niewinnie-wyglądające zadania nadpisujące sytuacje wyjątkowe które aktualnie są obsługiwane; to także redukuje często niechciane wydłużanie czasu życia przedmiotów do których odnosi się ramka stosu w wypisie śladu wywołań.

Jako nadrzędną zasadę, przyjmuje się że zadanie które wywołuje inne zadanie do wykonania pewnych operacji powinno sprawdzić czy wywołane zadanie zgłosiło wyjątek, a jeśli tak, to przekazać stan wyjątku do wywołującego. Powinno też odrzucić jakiekolwiek odniesienia do przedmiotów, które posiada, i zwrócić sygnalizator błędu, ale nie powinno ustawiać innego wyjątku — który nadpisywałby wyjątek, który właśnie został zgłoszony i tracić istotne informacje o dokładnym powodzie błędu.

A simple example of detecting exceptions and passing them on is shown in the `sum_sequence()` example above. It so happens that this example doesn't need to clean up any owned references when it detects an error. The following example function shows some error cleanup. First, to remind you why you like Python, we show the equivalent Python code:

```
def incr_item(dict, key):
    try:
        item = dict[key]
    except KeyError:
        item = 0
    dict[key] = item + 1
```

Tu następuje odpowiadający kod w języku C, w całej pełni okazałości:

```
int
incr_item(PyObject *dict, PyObject *key)
{
    /* Objects all initialized to NULL for Py_XDECREF */
    PyObject *item = NULL, *const_one = NULL, *incremented_item = NULL;
    int rv = -1; /* Return value initialized to -1 (failure) */

    item = PyObject_GetItem(dict, key);
    if (item == NULL) {
        /* Handle KeyError only: */
        if (!PyErr_ExceptionMatches(PyExc_KeyError))
            goto error;

        /* Clear the error and use zero: */
        PyErr_Clear();
        item = PyLong_FromLong(0L);
        if (item == NULL)
            goto error;
    }
    const_one = PyLong_FromLong(1L);
    if (const_one == NULL)
        goto error;
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
    incremented_item = PyNumber_Add(item, const_one);
    if (incremented_item == NULL)
        goto error;

    if (PyObject_SetItem(dict, key, incremented_item) < 0)
        goto error;
    rv = 0; /* Success */
    /* Continue with cleanup code */

 error:
    /* Cleanup code, shared by success and failure path */

    /* Use Py_XDECREF() to ignore NULL references */
    Py_XDECREF(item);
    Py_XDECREF(const_one);
    Py_XDECREF(incremented_item);

    return rv; /* -1 for error, 0 for success */
}
```

This example represents an endorsed use of the `goto` statement in C! It illustrates the use of `PyErr_ExceptionMatches()` and `PyErr_Clear()` to handle specific exceptions, and the use of `Py_XDECREF()` to dispose of owned references that may be `NULL` (note the `'X'` in the name; `Py_DECREF()` would crash when confronted with a `NULL` reference). It is important that the variables used to hold owned references are initialized to `NULL` for this to work; likewise, the proposed return value is initialized to `-1` (failure) and only set to success after the final call made is successful.

1.6 Załączanie programu interpretującego język pytonowski

Jedno istotne zadanie, o które załączający (w przeciwieństwie do piszących rozszerzenia) program interpretujący język pytonowski muszą się martwić jest zainicjowanie i prawdopodobne zakończenie programu interpretującego polecenia języka pytonowskiego. Większość użyteczności programu interpretującego polecenia języka pytonowskiego może tylko być użyta po jego zainicjowaniu.

The basic initialization function is `Py_Initialize()`. This initializes the table of loaded modules, and creates the fundamental modules `builtins`, `__main__`, and `sys`. It also initializes the module search path (`sys.path`).

`Py_Initialize()` does not set the „script argument list" (`sys.argv`). If this variable is needed by Python code that will be executed later, setting `PyConfig.argv` and `PyConfig.parse_argv` must be set: see *Python Initialization Configuration*.

On most systems (in particular, on Unix and Windows, although the details are slightly different), `Py_Initialize()` calculates the module search path based upon its best guess for the location of the standard Python interpreter executable, assuming that the Python library is found in a fixed location relative to the Python interpreter executable. In particular, it looks for a directory named `lib/pythonX.Y` relative to the parent directory where the executable named `python` is found on the shell command search path (the environment variable `PATH`).

Na przykład, jeśli plik wykonywalny programu interpretującego polecenia języka pytonowskiego znajduje się w katalogu `/usr/local/bin/python`, będzie zakładał, że biblioteki są w katalogu `/usr/local/lib/pythonX.Y` (Faktycznie, ta szczególna ścieżka jest także „ratunkowym" położeniem, używanym gdy żaden plik wykonywalny nazwany `python` nie znajdzie się w katalogach znajdujących się w zmiennej środowiskowej `PATH`.) Użytkownik może podmienić to zachowanie przez ustawienie zmiennej środowiskowej `PYTHONHOME`, lub wstawić dodatkowe katalogi przed sztandarową ścieżką przez ustawienie zmiennej środowiskowej `PYTHONPATH`.

The embedding application can steer the search by setting `PyConfig.program_name` *before* calling `Py_InitializeFromConfig()`. Note that `PYTHONHOME` still overrides this and `PYTHONPATH` is still inserted in front of the standard path. An application that requires total control has to provide its own implementation of `Py_GetPath()`, `Py_GetPrefix()`, `Py_GetExecPrefix()`, and `Py_GetProgramFullPath()` (all defined in `Modules/getpath.c`).

Sometimes, it is desirable to „uninitialize" Python. For instance, the application may want to start over (make another call to `Py_Initialize()`) or the application is simply done with its use of Python and wants to free memory allocated by Python. This can be accomplished by calling `Py_FinalizeEx()`. The function `Py_IsInitialized()` returns true if Python is currently in the initialized state. More information about these functions is given in a later chapter. Notice that `Py_FinalizeEx()` does *not* free all memory allocated by the Python interpreter, e.g. memory allocated by extension modules currently cannot be released.

1.7 Odpluskwiające Budowy

Program interpretujący język pytonowski może być zbudowany z kilkoma makropoleceniami do załączenia dodatkowych sprawdzeń programu interpretującego polecenia języka pytonowskiego i modułów rozszerzających. Te sprawdzenia mają zwyczaj dodawać duży narzut czasu wykonania poleceń programu więc nie są załączane domyślnie.

A full list of the various types of debugging builds is in the file `Misc/SpecialBuilds.txt` in the Python source distribution. Builds are available that support tracing of reference counts, debugging the memory allocator, or low--level profiling of the main interpreter loop. Only the most frequently used builds will be described in the remainder of this section.

`Py_DEBUG`

Compiling the interpreter with the `Py_DEBUG` macro defined produces what is generally meant by a debug build of Python. `Py_DEBUG` is enabled in the Unix build by adding `--with-pydebug` to the `./configure` command. It is also implied by the presence of the not-Python-specific `_DEBUG` macro. When `Py_DEBUG` is enabled in the Unix build, compiler optimization is disabled.

In addition to the reference count debugging described below, extra checks are performed, see Python Debug Build.

Defining `Py_TRACE_REFS` enables reference tracing (see the `configure --with-trace-refs option`). When defined, a circular doubly linked list of active objects is maintained by adding two extra fields to every `PyObject`. Total allocations are tracked as well. Upon exit, all existing references are printed. (In interactive mode this happens after every statement run by the interpreter.)

Odwołaj się do `Misc/SpecialBuilds.txt` w źródłowym pakiecie języka pytonowskiego po więcej szczegółów.

ROZDZIAŁ 2

C API Stability

Unless documented otherwise, Python's C API is covered by the Backwards Compatibility Policy, **PEP 387**. Most changes to it are source-compatible (typically by only adding new API). Changing existing API or removing API is only done after a deprecation period or to fix serious issues.

CPython's Application Binary Interface (ABI) is forward- and backwards-compatible across a minor release (if these are compiled the same way; see *Platform Considerations* below). So, code compiled for Python 3.10.0 will work on 3.10.8 and vice versa, but will need to be compiled separately for 3.9.x and 3.11.x.

There are two tiers of C API with different stability expectations:

- *Unstable API*, may change in minor versions without a deprecation period. It is marked by the `PyUnstable` prefix in names.
- *Limited API*, is compatible across several minor releases. When `Py_LIMITED_API` is defined, only this subset is exposed from `Python.h`.

These are discussed in more detail below.

Names prefixed by an underscore, such as `_Py_InternalState`, are private API that can change without notice even in patch releases. If you need to use this API, consider reaching out to CPython developers to discuss adding public API for your use case.

2.1 Unstable C API

Any API named with the `PyUnstable` prefix exposes CPython implementation details, and may change in every minor release (e.g. from 3.9 to 3.10) without any deprecation warnings. However, it will not change in a bugfix release (e.g. from 3.10.0 to 3.10.1).

It is generally intended for specialized, low-level tools like debuggers.

Projects that use this API are expected to follow CPython development and spend extra effort adjusting to changes.

2.2 Stable Application Binary Interface

For simplicity, this document talks about *extensions*, but the Limited API and Stable ABI work the same way for all uses of the API – for example, embedding Python.

2.2.1 Limited C API

Python 3.2 introduced the *Limited API*, a subset of Python's C API. Extensions that only use the Limited API can be compiled once and work with multiple versions of Python. Contents of the Limited API are *listed below*.

Py_LIMITED_API

Define this macro before including `Python.h` to opt in to only use the Limited API, and to select the Limited API version.

Define `Py_LIMITED_API` to the value of *`PY_VERSION_HEX`* corresponding to the lowest Python version your extension supports. The extension will work without recompilation with all Python 3 releases from the specified one onward, and can use Limited API introduced up to that version.

Rather than using the `PY_VERSION_HEX` macro directly, hardcode a minimum minor version (e.g. `0x030A0000` for Python 3.10) for stability when compiling with future Python versions.

You can also define `Py_LIMITED_API` to `3`. This works the same as `0x03020000` (Python 3.2, the version that introduced Limited API).

2.2.2 stabilnego ABI

To enable this, Python provides a *Stable ABI*: a set of symbols that will remain compatible across Python 3.x versions.

The Stable ABI contains symbols exposed in the *Limited API*, but also other ones – for example, functions necessary to support older versions of the Limited API.

On Windows, extensions that use the Stable ABI should be linked against `python3.dll` rather than a version-specific library such as `python39.dll`.

On some platforms, Python will look for and load shared library files named with the `abi3` tag (e.g. `mymodule.abi3.so`). It does not check if such extensions conform to a Stable ABI. The user (or their packaging tools) need to ensure that, for example, extensions built with the 3.10+ Limited API are not installed for lower versions of Python.

All functions in the Stable ABI are present as functions in Python's shared library, not solely as macros. This makes them usable from languages that don't use the C preprocessor.

2.2.3 Limited API Scope and Performance

The goal for the Limited API is to allow everything that is possible with the full C API, but possibly with a performance penalty.

For example, while *`PyList_GetItem()`* is available, its "unsafe" macro variant *`PyList_GET_ITEM()`* is not. The macro can be faster because it can rely on version-specific implementation details of the list object.

Without `Py_LIMITED_API` defined, some C API functions are inlined or replaced by macros. Defining `Py_LIMITED_API` disables this inlining, allowing stability as Python's data structures are improved, but possibly reducing performance.

By leaving out the `Py_LIMITED_API` definition, it is possible to compile a Limited API extension with a version-specific ABI. This can improve performance for that Python version, but will limit compatibility. Compiling with `Py_LIMITED_API` will then yield an extension that can be distributed where a version-specific one is not available – for example, for prereleases of an upcoming Python version.

2.2.4 Limited API Caveats

Note that compiling with `Py_LIMITED_API` is *not* a complete guarantee that code conforms to the *Limited API* or the *Stable ABI*. `Py_LIMITED_API` only covers definitions, but an API also includes other issues, such as expected semantics.

One issue that `Py_LIMITED_API` does not guard against is calling a function with arguments that are invalid in a lower Python version. For example, consider a function that starts accepting `NULL` for an argument. In Python 3.9, `NULL` now selects a default behavior, but in Python 3.8, the argument will be used directly, causing a `NULL` dereference and crash. A similar argument works for fields of structs.

Another issue is that some struct fields are currently not hidden when `Py_LIMITED_API` is defined, even though they're part of the Limited API.

For these reasons, we recommend testing an extension with *all* minor Python versions it supports, and preferably to build with the *lowest* such version.

We also recommend reviewing documentation of all used API to check if it is explicitly part of the Limited API. Even with `Py_LIMITED_API` defined, a few private declarations are exposed for technical reasons (or even unintentionally, as bugs).

Also note that the Limited API is not necessarily stable: compiling with `Py_LIMITED_API` with Python 3.8 means that the extension will run with Python 3.12, but it will not necessarily *compile* with Python 3.12. In particular, parts of the Limited API may be deprecated and removed, provided that the Stable ABI stays stable.

2.3 Platform Considerations

ABI stability depends not only on Python, but also on the compiler used, lower-level libraries and compiler options. For the purposes of the *Stable ABI*, these details define a "platform". They usually depend on the OS type and processor architecture

It is the responsibility of each particular distributor of Python to ensure that all Python versions on a particular platform are built in a way that does not break the Stable ABI. This is the case with Windows and macOS releases from `python.org` and many third-party distributors.

2.4 Contents of Limited API

Currently, the *Limited API* includes the following items:

- *PY_VECTORCALL_ARGUMENTS_OFFSET*
- *PyAIter_Check()*
- *PyArg_Parse()*
- *PyArg_ParseTuple()*
- *PyArg_ParseTupleAndKeywords()*
- *PyArg_UnpackTuple()*
- *PyArg_VaParse()*
- *PyArg_VaParseTupleAndKeywords()*
- *PyArg_ValidateKeywordArguments()*
- PyBaseObject_Type
- *PyBool_FromLong()*
- *PyBool_Type*
- *PyBuffer_FillContiguousStrides()*
- *PyBuffer_FillInfo()*
- *PyBuffer_FromContiguous()*
- *PyBuffer_GetPointer()*
- *PyBuffer_IsContiguous()*
- *PyBuffer_Release()*
- *PyBuffer_SizeFromFormat()*
- *PyBuffer_ToContiguous()*

- PyByteArrayIter_Type
- *PyByteArray_AsString()*
- *PyByteArray_Concat()*
- *PyByteArray_FromObject()*
- *PyByteArray_FromStringAndSize()*
- *PyByteArray_Resize()*
- *PyByteArray_Size()*
- *PyByteArray_Type*
- PyBytesIter_Type
- *PyBytes_AsString()*
- *PyBytes_AsStringAndSize()*
- *PyBytes_Concat()*
- *PyBytes_ConcatAndDel()*
- PyBytes_DecodeEscape()
- *PyBytes_FromFormat()*
- *PyBytes_FromFormatV()*
- *PyBytes_FromObject()*
- *PyBytes_FromString()*
- *PyBytes_FromStringAndSize()*
- PyBytes_Repr()
- *PyBytes_Size()*
- *PyBytes_Type*
- *PyCFunction*
- *PyCFunctionFast*
- *PyCFunctionFastWithKeywords*
- *PyCFunctionWithKeywords*
- PyCFunction_GetFlags()
- PyCFunction_GetFunction()
- PyCFunction_GetSelf()
- *PyCFunction_New()*
- *PyCFunction_NewEx()*
- PyCFunction_Type
- *PyCMethod_New()*
- *PyCallIter_New()*
- *PyCallIter_Type*
- *PyCallable_Check()*
- *PyCapsule_Destructor*
- *PyCapsule_GetContext()*
- *PyCapsule_GetDestructor()*

- *PyCapsule_GetName()*
- *PyCapsule_GetPointer()*
- *PyCapsule_Import()*
- *PyCapsule_IsValid()*
- *PyCapsule_New()*
- *PyCapsule_SetContext()*
- *PyCapsule_SetDestructor()*
- *PyCapsule_SetName()*
- *PyCapsule_SetPointer()*
- PyCapsule_Type
- PyClassMethodDescr_Type
- *PyCodec_BackslashReplaceErrors()*
- *PyCodec_Decode()*
- *PyCodec_Decoder()*
- *PyCodec_Encode()*
- *PyCodec_Encoder()*
- *PyCodec_IgnoreErrors()*
- *PyCodec_IncrementalDecoder()*
- *PyCodec_IncrementalEncoder()*
- *PyCodec_KnownEncoding()*
- *PyCodec_LookupError()*
- *PyCodec_NameReplaceErrors()*
- *PyCodec_Register()*
- *PyCodec_RegisterError()*
- *PyCodec_ReplaceErrors()*
- *PyCodec_StreamReader()*
- *PyCodec_StreamWriter()*
- *PyCodec_StrictErrors()*
- *PyCodec_Unregister()*
- *PyCodec_XMLCharRefReplaceErrors()*
- *PyComplex_FromDoubles()*
- *PyComplex_ImagAsDouble()*
- *PyComplex_RealAsDouble()*
- *PyComplex_Type*
- *PyDescr_NewClassMethod()*
- *PyDescr_NewGetSet()*
- *PyDescr_NewMember()*
- *PyDescr_NewMethod()*
- PyDictItems_Type

- PyDictIterItem_Type
- PyDictIterKey_Type
- PyDictIterValue_Type
- PyDictKeys_Type
- *PyDictProxy_New()*
- PyDictProxy_Type
- PyDictRevIterItem_Type
- PyDictRevIterKey_Type
- PyDictRevIterValue_Type
- PyDictValues_Type
- *PyDict_Clear()*
- *PyDict_Contains()*
- *PyDict_Copy()*
- *PyDict_DelItem()*
- *PyDict_DelItemString()*
- *PyDict_GetItem()*
- *PyDict_GetItemRef()*
- *PyDict_GetItemString()*
- *PyDict_GetItemStringRef()*
- *PyDict_GetItemWithError()*
- *PyDict_Items()*
- *PyDict_Keys()*
- *PyDict_Merge()*
- *PyDict_MergeFromSeq2()*
- *PyDict_New()*
- *PyDict_Next()*
- *PyDict_SetItem()*
- *PyDict_SetItemString()*
- *PyDict_Size()*
- *PyDict_Type*
- *PyDict_Update()*
- *PyDict_Values()*
- PyEllipsis_Type
- PyEnum_Type
- *PyErr_BadArgument()*
- *PyErr_BadInternalCall()*
- *PyErr_CheckSignals()*
- *PyErr_Clear()*
- PyErr_Display()

- *PyErr_DisplayException()*
- *PyErr_ExceptionMatches()*
- *PyErr_Fetch()*
- *PyErr_Format()*
- *PyErr_FormatV()*
- *PyErr_GetExcInfo()*
- *PyErr_GetHandledException()*
- *PyErr_GetRaisedException()*
- *PyErr_GivenExceptionMatches()*
- *PyErr_NewException()*
- *PyErr_NewExceptionWithDoc()*
- *PyErr_NoMemory()*
- *PyErr_NormalizeException()*
- *PyErr_Occurred()*
- *PyErr_Print()*
- *PyErr_PrintEx()*
- PyErr_ProgramText()
- *PyErr_ResourceWarning()*
- *PyErr_Restore()*
- *PyErr_SetExcFromWindowsErr()*
- *PyErr_SetExcFromWindowsErrWithFilename()*
- *PyErr_SetExcFromWindowsErrWithFilenameObject()*
- *PyErr_SetExcFromWindowsErrWithFilenameObjects()*
- *PyErr_SetExcInfo()*
- *PyErr_SetFromErrno()*
- *PyErr_SetFromErrnoWithFilename()*
- *PyErr_SetFromErrnoWithFilenameObject()*
- *PyErr_SetFromErrnoWithFilenameObjects()*
- *PyErr_SetFromWindowsErr()*
- *PyErr_SetFromWindowsErrWithFilename()*
- *PyErr_SetHandledException()*
- *PyErr_SetImportError()*
- *PyErr_SetImportErrorSubclass()*
- *PyErr_SetInterrupt()*
- *PyErr_SetInterruptEx()*
- *PyErr_SetNone()*
- *PyErr_SetObject()*
- *PyErr_SetRaisedException()*
- *PyErr_SetString()*

- *PyErr_SyntaxLocation()*
- *PyErr_SyntaxLocationEx()*
- *PyErr_WarnEx()*
- *PyErr_WarnExplicit()*
- *PyErr_WarnFormat()*
- *PyErr_WriteUnraisable()*
- *PyEval_AcquireThread()*
- *PyEval_EvalCode()*
- *PyEval_EvalCodeEx()*
- *PyEval_EvalFrame()*
- *PyEval_EvalFrameEx()*
- *PyEval_GetBuiltins()*
- *PyEval_GetFrame()*
- *PyEval_GetFrameBuiltins()*
- *PyEval_GetFrameGlobals()*
- *PyEval_GetFrameLocals()*
- *PyEval_GetFuncDesc()*
- *PyEval_GetFuncName()*
- *PyEval_GetGlobals()*
- *PyEval_GetLocals()*
- *PyEval_InitThreads()*
- *PyEval_ReleaseThread()*
- *PyEval_RestoreThread()*
- *PyEval_SaveThread()*
- PyExc_ArithmeticError
- PyExc_AssertionError
- PyExc_AttributeError
- PyExc_BaseException
- PyExc_BaseExceptionGroup
- PyExc_BlockingIOError
- PyExc_BrokenPipeError
- PyExc_BufferError
- PyExc_BytesWarning
- PyExc_ChildProcessError
- PyExc_ConnectionAbortedError
- PyExc_ConnectionError
- PyExc_ConnectionRefusedError
- PyExc_ConnectionResetError
- PyExc_DeprecationWarning

- PyExc_EOFError
- PyExc_EncodingWarning
- PyExc_EnvironmentError
- PyExc_Exception
- PyExc_FileExistsError
- PyExc_FileNotFoundError
- PyExc_FloatingPointError
- PyExc_FutureWarning
- PyExc_GeneratorExit
- PyExc_IOError
- PyExc_ImportError
- PyExc_ImportWarning
- PyExc_IndentationError
- PyExc_IndexError
- PyExc_InterruptedError
- PyExc_IsADirectoryError
- PyExc_KeyError
- PyExc_KeyboardInterrupt
- PyExc_LookupError
- PyExc_MemoryError
- PyExc_ModuleNotFoundError
- PyExc_NameError
- PyExc_NotADirectoryError
- PyExc_NotImplementedError
- PyExc_OSError
- PyExc_OverflowError
- PyExc_PendingDeprecationWarning
- PyExc_PermissionError
- PyExc_ProcessLookupError
- PyExc_RecursionError
- PyExc_ReferenceError
- PyExc_ResourceWarning
- PyExc_RuntimeError
- PyExc_RuntimeWarning
- PyExc_StopAsyncIteration
- PyExc_StopIteration
- PyExc_SyntaxError
- PyExc_SyntaxWarning
- PyExc_SystemError

- PyExc_SystemExit
- PyExc_TabError
- PyExc_TimeoutError
- PyExc_TypeError
- PyExc_UnboundLocalError
- PyExc_UnicodeDecodeError
- PyExc_UnicodeEncodeError
- PyExc_UnicodeError
- PyExc_UnicodeTranslateError
- PyExc_UnicodeWarning
- PyExc_UserWarning
- PyExc_ValueError
- PyExc_Warning
- PyExc_WindowsError
- PyExc_ZeroDivisionError
- PyExceptionClass_Name()
- *PyException_GetArgs()*
- *PyException_GetCause()*
- *PyException_GetContext()*
- *PyException_GetTraceback()*
- *PyException_SetArgs()*
- *PyException_SetCause()*
- *PyException_SetContext()*
- *PyException_SetTraceback()*
- *PyFile_FromFd()*
- *PyFile_GetLine()*
- *PyFile_WriteObject()*
- *PyFile_WriteString()*
- PyFilter_Type
- *PyFloat_AsDouble()*
- *PyFloat_FromDouble()*
- *PyFloat_FromString()*
- *PyFloat_GetInfo()*
- *PyFloat_GetMax()*
- *PyFloat_GetMin()*
- *PyFloat_Type*
- *PyFrameObject*
- *PyFrame_GetCode()*
- *PyFrame_GetLineNumber()*

- *PyFrozenSet_New()*
- *PyFrozenSet_Type*
- *PyGC_Collect()*
- *PyGC_Disable()*
- *PyGC_Enable()*
- *PyGC_IsEnabled()*
- *PyGILState_Ensure()*
- *PyGILState_GetThisThreadState()*
- *PyGILState_Release()*
- PyGILState_STATE
- *PyGetSetDef*
- PyGetSetDescr_Type
- *PyImport_AddModule()*
- *PyImport_AddModuleObject()*
- *PyImport_AddModuleRef()*
- *PyImport_AppendInittab()*
- *PyImport_ExecCodeModule()*
- *PyImport_ExecCodeModuleEx()*
- *PyImport_ExecCodeModuleObject()*
- *PyImport_ExecCodeModuleWithPathnames()*
- *PyImport_GetImporter()*
- *PyImport_GetMagicNumber()*
- *PyImport_GetMagicTag()*
- *PyImport_GetModule()*
- *PyImport_GetModuleDict()*
- *PyImport_Import()*
- *PyImport_ImportFrozenModule()*
- *PyImport_ImportFrozenModuleObject()*
- *PyImport_ImportModule()*
- *PyImport_ImportModuleLevel()*
- *PyImport_ImportModuleLevelObject()*
- *PyImport_ImportModuleNoBlock()*
- *PyImport_ReloadModule()*
- *PyIndex_Check()*
- *PyInterpreterState*
- *PyInterpreterState_Clear()*
- *PyInterpreterState_Delete()*
- *PyInterpreterState_Get()*
- *PyInterpreterState_GetDict()*

- *PyInterpreterState_GetID()*
- *PyInterpreterState_New()*
- *PyIter_Check()*
- *PyIter_Next()*
- *PyIter_Send()*
- PyListIter_Type
- PyListRevIter_Type
- *PyList_Append()*
- *PyList_AsTuple()*
- *PyList_GetItem()*
- *PyList_GetItemRef()*
- *PyList_GetSlice()*
- *PyList_Insert()*
- *PyList_New()*
- *PyList_Reverse()*
- *PyList_SetItem()*
- *PyList_SetSlice()*
- *PyList_Size()*
- *PyList_Sort()*
- *PyList_Type*
- *PyLongObject*
- PyLongRangeIter_Type
- *PyLong_AsDouble()*
- *PyLong_AsInt()*
- *PyLong_AsLong()*
- *PyLong_AsLongAndOverflow()*
- *PyLong_AsLongLong()*
- *PyLong_AsLongLongAndOverflow()*
- *PyLong_AsSize_t()*
- *PyLong_AsSsize_t()*
- *PyLong_AsUnsignedLong()*
- *PyLong_AsUnsignedLongLong()*
- *PyLong_AsUnsignedLongLongMask()*
- *PyLong_AsUnsignedLongMask()*
- *PyLong_AsVoidPtr()*
- *PyLong_FromDouble()*
- *PyLong_FromLong()*
- *PyLong_FromLongLong()*
- *PyLong_FromSize_t()*

- *PyLong_FromSsize_t()*
- *PyLong_FromString()*
- *PyLong_FromUnsignedLong()*
- *PyLong_FromUnsignedLongLong()*
- *PyLong_FromVoidPtr()*
- *PyLong_GetInfo()*
- *PyLong_Type*
- PyMap_Type
- *PyMapping_Check()*
- *PyMapping_GetItemString()*
- *PyMapping_GetOptionalItem()*
- *PyMapping_GetOptionalItemString()*
- *PyMapping_HasKey()*
- *PyMapping_HasKeyString()*
- *PyMapping_HasKeyStringWithError()*
- *PyMapping_HasKeyWithError()*
- *PyMapping_Items()*
- *PyMapping_Keys()*
- *PyMapping_Length()*
- *PyMapping_SetItemString()*
- *PyMapping_Size()*
- *PyMapping_Values()*
- *PyMem_Calloc()*
- *PyMem_Free()*
- *PyMem_Malloc()*
- *PyMem_RawCalloc()*
- *PyMem_RawFree()*
- *PyMem_RawMalloc()*
- *PyMem_RawRealloc()*
- *PyMem_Realloc()*
- *PyMemberDef*
- PyMemberDescr_Type
- *PyMember_GetOne()*
- *PyMember_SetOne()*
- *PyMemoryView_FromBuffer()*
- *PyMemoryView_FromMemory()*
- *PyMemoryView_FromObject()*
- *PyMemoryView_GetContiguous()*
- PyMemoryView_Type

- *PyMethodDef*
- PyMethodDescr_Type
- *PyModuleDef*
- PyModuleDef_Base
- *PyModuleDef_Init()*
- PyModuleDef_Type
- *PyModule_Add()*
- *PyModule_AddFunctions()*
- *PyModule_AddIntConstant()*
- *PyModule_AddObject()*
- *PyModule_AddObjectRef()*
- *PyModule_AddStringConstant()*
- *PyModule_AddType()*
- *PyModule_Create2()*
- *PyModule_ExecDef()*
- *PyModule_FromDefAndSpec2()*
- *PyModule_GetDef()*
- *PyModule_GetDict()*
- *PyModule_GetFilename()*
- *PyModule_GetFilenameObject()*
- *PyModule_GetName()*
- *PyModule_GetNameObject()*
- *PyModule_GetState()*
- *PyModule_New()*
- *PyModule_NewObject()*
- *PyModule_SetDocString()*
- *PyModule_Type*
- *PyNumber_Absolute()*
- *PyNumber_Add()*
- *PyNumber_And()*
- *PyNumber_AsSsize_t()*
- *PyNumber_Check()*
- *PyNumber_Divmod()*
- *PyNumber_Float()*
- *PyNumber_FloorDivide()*
- *PyNumber_InPlaceAdd()*
- *PyNumber_InPlaceAnd()*
- *PyNumber_InPlaceFloorDivide()*
- *PyNumber_InPlaceLshift()*

- *PyNumber_InPlaceMatrixMultiply()*
- *PyNumber_InPlaceMultiply()*
- *PyNumber_InPlaceOr()*
- *PyNumber_InPlacePower()*
- *PyNumber_InPlaceRemainder()*
- *PyNumber_InPlaceRshift()*
- *PyNumber_InPlaceSubtract()*
- *PyNumber_InPlaceTrueDivide()*
- *PyNumber_InPlaceXor()*
- *PyNumber_Index()*
- *PyNumber_Invert()*
- *PyNumber_Long()*
- *PyNumber_Lshift()*
- *PyNumber_MatrixMultiply()*
- *PyNumber_Multiply()*
- *PyNumber_Negative()*
- *PyNumber_Or()*
- *PyNumber_Positive()*
- *PyNumber_Power()*
- *PyNumber_Remainder()*
- *PyNumber_Rshift()*
- *PyNumber_Subtract()*
- *PyNumber_ToBase()*
- *PyNumber_TrueDivide()*
- *PyNumber_Xor()*
- *PyOS_AfterFork()*
- *PyOS_AfterFork_Child()*
- *PyOS_AfterFork_Parent()*
- *PyOS_BeforeFork()*
- *PyOS_CheckStack()*
- *PyOS_FSPath()*
- *PyOS_InputHook*
- PyOS_InterruptOccurred()
- *PyOS_double_to_string()*
- *PyOS_getsig()*
- PyOS_mystricmp()
- PyOS_mystrnicmp()
- *PyOS_setsig()*
- *PyOS_sighandler_t*

- PyOS_snprintf()
- PyOS_string_to_double()
- PyOS_strtol()
- PyOS_strtoul()
- PyOS_vsnprintf()
- PyObject
- PyObject.ob_refcnt
- PyObject.ob_type
- PyObject_ASCII()
- PyObject_AsFileDescriptor()
- PyObject_Bytes()
- PyObject_Call()
- PyObject_CallFunction()
- PyObject_CallFunctionObjArgs()
- PyObject_CallMethod()
- PyObject_CallMethodObjArgs()
- PyObject_CallNoArgs()
- PyObject_CallObject()
- PyObject_Calloc()
- PyObject_CheckBuffer()
- PyObject_ClearWeakRefs()
- PyObject_CopyData()
- PyObject_DelAttr()
- PyObject_DelAttrString()
- PyObject_DelItem()
- PyObject_DelItemString()
- PyObject_Dir()
- PyObject_Format()
- PyObject_Free()
- PyObject_GC_Del()
- PyObject_GC_IsFinalized()
- PyObject_GC_IsTracked()
- PyObject_GC_Track()
- PyObject_GC_UnTrack()
- PyObject_GenericGetAttr()
- PyObject_GenericGetDict()
- PyObject_GenericSetAttr()
- PyObject_GenericSetDict()
- PyObject_GetAIter()

- *PyObject_GetAttr()*
- *PyObject_GetAttrString()*
- *PyObject_GetBuffer()*
- *PyObject_GetItem()*
- *PyObject_GetIter()*
- *PyObject_GetOptionalAttr()*
- *PyObject_GetOptionalAttrString()*
- *PyObject_GetTypeData()*
- *PyObject_HasAttr()*
- *PyObject_HasAttrString()*
- *PyObject_HasAttrStringWithError()*
- *PyObject_HasAttrWithError()*
- *PyObject_Hash()*
- *PyObject_HashNotImplemented()*
- *PyObject_Init()*
- *PyObject_InitVar()*
- *PyObject_IsInstance()*
- *PyObject_IsSubclass()*
- *PyObject_IsTrue()*
- *PyObject_Length()*
- *PyObject_Malloc()*
- *PyObject_Not()*
- *PyObject_Realloc()*
- *PyObject_Repr()*
- *PyObject_RichCompare()*
- *PyObject_RichCompareBool()*
- PyObject_SelfIter()
- *PyObject_SetAttr()*
- *PyObject_SetAttrString()*
- *PyObject_SetItem()*
- *PyObject_Size()*
- *PyObject_Str()*
- *PyObject_Type()*
- *PyObject_Vectorcall()*
- *PyObject_VectorcallMethod()*
- *PyProperty_Type*
- PyRangeIter_Type
- PyRange_Type
- PyReversed_Type

- *PySeqIter_New()*
- *PySeqIter_Type*
- *PySequence_Check()*
- *PySequence_Concat()*
- *PySequence_Contains()*
- *PySequence_Count()*
- *PySequence_DelItem()*
- *PySequence_DelSlice()*
- *PySequence_Fast()*
- *PySequence_GetItem()*
- *PySequence_GetSlice()*
- PySequence_In()
- *PySequence_InPlaceConcat()*
- *PySequence_InPlaceRepeat()*
- *PySequence_Index()*
- *PySequence_Length()*
- *PySequence_List()*
- *PySequence_Repeat()*
- *PySequence_SetItem()*
- *PySequence_SetSlice()*
- *PySequence_Size()*
- *PySequence_Tuple()*
- PySetIter_Type
- *PySet_Add()*
- *PySet_Clear()*
- *PySet_Contains()*
- *PySet_Discard()*
- *PySet_New()*
- *PySet_Pop()*
- *PySet_Size()*
- *PySet_Type*
- *PySlice_AdjustIndices()*
- *PySlice_GetIndices()*
- *PySlice_GetIndicesEx()*
- *PySlice_New()*
- *PySlice_Type*
- *PySlice_Unpack()*
- *PyState_AddModule()*
- *PyState_FindModule()*

- *PyState_RemoveModule()*
- *PyStructSequence_Desc*
- *PyStructSequence_Field*
- *PyStructSequence_GetItem()*
- *PyStructSequence_New()*
- *PyStructSequence_NewType()*
- *PyStructSequence_SetItem()*
- *PyStructSequence_UnnamedField*
- PySuper_Type
- *PySys_Audit()*
- *PySys_AuditTuple()*
- *PySys_FormatStderr()*
- *PySys_FormatStdout()*
- *PySys_GetObject()*
- *PySys_GetXOptions()*
- *PySys_ResetWarnOptions()*
- *PySys_SetArgv()*
- *PySys_SetArgvEx()*
- *PySys_SetObject()*
- *PySys_WriteStderr()*
- *PySys_WriteStdout()*
- *PyThreadState*
- *PyThreadState_Clear()*
- *PyThreadState_Delete()*
- *PyThreadState_Get()*
- *PyThreadState_GetDict()*
- *PyThreadState_GetFrame()*
- *PyThreadState_GetID()*
- *PyThreadState_GetInterpreter()*
- *PyThreadState_New()*
- *PyThreadState_SetAsyncExc()*
- *PyThreadState_Swap()*
- PyThread_GetInfo()
- *PyThread_ReInitTLS()*
- PyThread_acquire_lock()
- PyThread_acquire_lock_timed()
- PyThread_allocate_lock()
- *PyThread_create_key()*
- *PyThread_delete_key()*

- *PyThread_delete_key_value()*
- PyThread_exit_thread()
- PyThread_free_lock()
- *PyThread_get_key_value()*
- PyThread_get_stacksize()
- PyThread_get_thread_ident()
- PyThread_get_thread_native_id()
- PyThread_init_thread()
- PyThread_release_lock()
- *PyThread_set_key_value()*
- PyThread_set_stacksize()
- PyThread_start_new_thread()
- *PyThread_tss_alloc()*
- *PyThread_tss_create()*
- *PyThread_tss_delete()*
- *PyThread_tss_free()*
- *PyThread_tss_get()*
- *PyThread_tss_is_created()*
- *PyThread_tss_set()*
- PyTraceBack_Here()
- PyTraceBack_Print()
- PyTraceBack_Type
- PyTupleIter_Type
- *PyTuple_GetItem()*
- *PyTuple_GetSlice()*
- *PyTuple_New()*
- *PyTuple_Pack()*
- *PyTuple_SetItem()*
- *PyTuple_Size()*
- *PyTuple_Type*
- *PyTypeObject*
- *PyType_ClearCache()*
- *PyType_FromMetaclass()*
- *PyType_FromModuleAndSpec()*
- *PyType_FromSpec()*
- *PyType_FromSpecWithBases()*
- *PyType_GenericAlloc()*
- *PyType_GenericNew()*
- *PyType_GetFlags()*

- *PyType_GetFullyQualifiedName()*
- *PyType_GetModule()*
- *PyType_GetModuleByDef()*
- *PyType_GetModuleName()*
- *PyType_GetModuleState()*
- *PyType_GetName()*
- *PyType_GetQualName()*
- *PyType_GetSlot()*
- *PyType_GetTypeDataSize()*
- *PyType_IsSubtype()*
- *PyType_Modified()*
- *PyType_Ready()*
- *PyType_Slot*
- *PyType_Spec*
- *PyType_Type*
- *PyUnicodeDecodeError_Create()*
- *PyUnicodeDecodeError_GetEncoding()*
- *PyUnicodeDecodeError_GetEnd()*
- *PyUnicodeDecodeError_GetObject()*
- *PyUnicodeDecodeError_GetReason()*
- *PyUnicodeDecodeError_GetStart()*
- *PyUnicodeDecodeError_SetEnd()*
- *PyUnicodeDecodeError_SetReason()*
- *PyUnicodeDecodeError_SetStart()*
- *PyUnicodeEncodeError_GetEncoding()*
- *PyUnicodeEncodeError_GetEnd()*
- *PyUnicodeEncodeError_GetObject()*
- *PyUnicodeEncodeError_GetReason()*
- *PyUnicodeEncodeError_GetStart()*
- *PyUnicodeEncodeError_SetEnd()*
- *PyUnicodeEncodeError_SetReason()*
- *PyUnicodeEncodeError_SetStart()*
- PyUnicodeIter_Type
- *PyUnicodeTranslateError_GetEnd()*
- *PyUnicodeTranslateError_GetObject()*
- *PyUnicodeTranslateError_GetReason()*
- *PyUnicodeTranslateError_GetStart()*
- *PyUnicodeTranslateError_SetEnd()*
- *PyUnicodeTranslateError_SetReason()*

- *PyUnicodeTranslateError_SetStart()*
- PyUnicode_Append()
- PyUnicode_AppendAndDel()
- *PyUnicode_AsASCIIString()*
- *PyUnicode_AsCharmapString()*
- PyUnicode_AsDecodedObject()
- PyUnicode_AsDecodedUnicode()
- PyUnicode_AsEncodedObject()
- *PyUnicode_AsEncodedString()*
- PyUnicode_AsEncodedUnicode()
- *PyUnicode_AsLatin1String()*
- *PyUnicode_AsMBCSString()*
- *PyUnicode_AsRawUnicodeEscapeString()*
- *PyUnicode_AsUCS4()*
- *PyUnicode_AsUCS4Copy()*
- *PyUnicode_AsUTF16String()*
- *PyUnicode_AsUTF32String()*
- *PyUnicode_AsUTF8AndSize()*
- *PyUnicode_AsUTF8String()*
- *PyUnicode_AsUnicodeEscapeString()*
- *PyUnicode_AsWideChar()*
- *PyUnicode_AsWideCharString()*
- PyUnicode_BuildEncodingMap()
- *PyUnicode_Compare()*
- *PyUnicode_CompareWithASCIIString()*
- *PyUnicode_Concat()*
- *PyUnicode_Contains()*
- *PyUnicode_Count()*
- *PyUnicode_Decode()*
- *PyUnicode_DecodeASCII()*
- *PyUnicode_DecodeCharmap()*
- PyUnicode_DecodeCodePageStateful()
- *PyUnicode_DecodeFSDefault()*
- *PyUnicode_DecodeFSDefaultAndSize()*
- *PyUnicode_DecodeLatin1()*
- *PyUnicode_DecodeLocale()*
- *PyUnicode_DecodeLocaleAndSize()*
- *PyUnicode_DecodeMBCS()*
- *PyUnicode_DecodeMBCSStateful()*

- *PyUnicode_DecodeRawUnicodeEscape()*
- *PyUnicode_DecodeUTF16()*
- *PyUnicode_DecodeUTF16Stateful()*
- *PyUnicode_DecodeUTF32()*
- *PyUnicode_DecodeUTF32Stateful()*
- *PyUnicode_DecodeUTF7()*
- *PyUnicode_DecodeUTF7Stateful()*
- *PyUnicode_DecodeUTF8()*
- *PyUnicode_DecodeUTF8Stateful()*
- *PyUnicode_DecodeUnicodeEscape()*
- *PyUnicode_EncodeCodePage()*
- *PyUnicode_EncodeFSDefault()*
- *PyUnicode_EncodeLocale()*
- *PyUnicode_EqualToUTF8()*
- *PyUnicode_EqualToUTF8AndSize()*
- *PyUnicode_FSConverter()*
- *PyUnicode_FSDecoder()*
- *PyUnicode_Find()*
- *PyUnicode_FindChar()*
- *PyUnicode_Format()*
- *PyUnicode_FromEncodedObject()*
- *PyUnicode_FromFormat()*
- *PyUnicode_FromFormatV()*
- *PyUnicode_FromObject()*
- PyUnicode_FromOrdinal()
- *PyUnicode_FromString()*
- *PyUnicode_FromStringAndSize()*
- *PyUnicode_FromWideChar()*
- PyUnicode_GetDefaultEncoding()
- *PyUnicode_GetLength()*
- *PyUnicode_InternFromString()*
- *PyUnicode_InternInPlace()*
- *PyUnicode_IsIdentifier()*
- *PyUnicode_Join()*
- PyUnicode_Partition()
- PyUnicode_RPartition()
- PyUnicode_RSplit()
- *PyUnicode_ReadChar()*
- *PyUnicode_Replace()*

- PyUnicode_Resize()
- *PyUnicode_RichCompare()*
- *PyUnicode_Split()*
- *PyUnicode_Splitlines()*
- *PyUnicode_Substring()*
- *PyUnicode_Tailmatch()*
- *PyUnicode_Translate()*
- *PyUnicode_Type*
- *PyUnicode_WriteChar()*
- *PyVarObject*
- PyVarObject.ob_base
- *PyVarObject.ob_size*
- *PyVectorcall_Call()*
- *PyVectorcall_NARGS()*
- PyWeakReference
- *PyWeakref_GetObject()*
- *PyWeakref_GetRef()*
- *PyWeakref_NewProxy()*
- *PyWeakref_NewRef()*
- PyWrapperDescr_Type
- *PyWrapper_New()*
- PyZip_Type
- *Py_AddPendingCall()*
- *Py_AtExit()*
- *Py_BEGIN_ALLOW_THREADS*
- *Py_BLOCK_THREADS*
- *Py_BuildValue()*
- *Py_BytesMain()*
- *Py_CompileString()*
- *Py_DecRef()*
- *Py_DecodeLocale()*
- *Py_END_ALLOW_THREADS*
- *Py_EncodeLocale()*
- *Py_EndInterpreter()*
- *Py_EnterRecursiveCall()*
- *Py_Exit()*
- *Py_FatalError()*
- Py_FileSystemDefaultEncodeErrors
- Py_FileSystemDefaultEncoding

- *Py_Finalize()*
- *Py_FinalizeEx()*
- *Py_GenericAlias()*
- *Py_GenericAliasType*
- *Py_GetBuildInfo()*
- *Py_GetCompiler()*
- *Py_GetConstant()*
- *Py_GetConstantBorrowed()*
- *Py_GetCopyright()*
- *Py_GetExecPrefix()*
- *Py_GetPath()*
- *Py_GetPlatform()*
- *Py_GetPrefix()*
- *Py_GetProgramFullPath()*
- *Py_GetProgramName()*
- *Py_GetPythonHome()*
- Py_GetRecursionLimit()
- *Py_GetVersion()*
- Py_HasFileSystemDefaultEncoding
- *Py_IncRef()*
- *Py_Initialize()*
- *Py_InitializeEx()*
- *Py_Is()*
- *Py_IsFalse()*
- *Py_IsFinalizing()*
- *Py_IsInitialized()*
- *Py_IsNone()*
- *Py_IsTrue()*
- *Py_LeaveRecursiveCall()*
- *Py_Main()*
- Py_MakePendingCalls()
- *Py_NewInterpreter()*
- *Py_NewRef()*
- *Py_ReprEnter()*
- *Py_ReprLeave()*
- *Py_SetProgramName()*
- *Py_SetPythonHome()*
- Py_SetRecursionLimit()
- *Py_UCS4*

- *Py_UNBLOCK_THREADS*
- Py_UTF8Mode
- *Py_VaBuildValue()*
- *Py_Version*
- *Py_XNewRef()*
- *Py_buffer*
- Py_intptr_t
- *Py_ssize_t*
- Py_uintptr_t
- *allocfunc*
- *binaryfunc*
- *descrgetfunc*
- *descrsetfunc*
- *destructor*
- *getattrfunc*
- *getattrofunc*
- *getbufferproc*
- *getiterfunc*
- *getter*
- *hashfunc*
- *initproc*
- *inquiry*
- *iternextfunc*
- *lenfunc*
- *newfunc*
- *objobjargproc*
- *objobjproc*
- *releasebufferproc*
- *reprfunc*
- *richcmpfunc*
- *setattrfunc*
- *setattrofunc*
- *setter*
- *ssizeargfunc*
- *ssizeobjargproc*
- ssizessizeargfunc
- ssizessizeobjargproc
- symtable
- *ternaryfunc*

- *traverseproc*
- *unaryfunc*
- *vectorcallfunc*
- *visitproc*

ROZDZIAŁ 3

The Very High Level Layer

The functions in this chapter will let you execute Python source code given in a file or a buffer, but they will not let you interact in a more detailed way with the interpreter.

Several of these functions accept a start symbol from the grammar as a parameter. The available start symbols are `Py_eval_input`, `Py_file_input`, and `Py_single_input`. These are described following the functions which accept them as parameters.

Note also that several of these functions take `FILE*` parameters. One particular issue which needs to be handled carefully is that the `FILE` structure for different C libraries can be different and incompatible. Under Windows (at least), it is possible for dynamically linked extensions to actually use different libraries, so care should be taken that `FILE*` parameters are only passed to these functions if it is certain that they were created by the same library that the Python runtime is using.

int **PyRun_AnyFile**(FILE *fp, const char *filename)

This is a simplified interface to `PyRun_AnyFileExFlags()` below, leaving *closeit* set to `0` and *flags* set to `NULL`.

int **PyRun_AnyFileFlags**(FILE *fp, const char *filename, *PyCompilerFlags* *flags)

This is a simplified interface to `PyRun_AnyFileExFlags()` below, leaving the *closeit* argument set to `0`.

int **PyRun_AnyFileEx**(FILE *fp, const char *filename, int closeit)

This is a simplified interface to `PyRun_AnyFileExFlags()` below, leaving the *flags* argument set to `NULL`.

int **PyRun_AnyFileExFlags**(FILE *fp, const char *filename, int closeit, *PyCompilerFlags* *flags)

If *fp* refers to a file associated with an interactive device (console or terminal input or Unix pseudo-terminal), return the value of `PyRun_InteractiveLoop()`, otherwise return the result of `PyRun_SimpleFile()`. *filename* is decoded from the filesystem encoding (`sys.getfilesystemencoding()`). If *filename* is `NULL`, this function uses `"???"` as the filename. If *closeit* is true, the file is closed before `PyRun_SimpleFileExFlags()` returns.

int **PyRun_SimpleString**(const char *command)

This is a simplified interface to `PyRun_SimpleStringFlags()` below, leaving the `PyCompilerFlags*` argument set to `NULL`.

int **PyRun_SimpleStringFlags**(const char *command, *PyCompilerFlags* *flags)

Executes the Python source code from *command* in the `__main__` module according to the *flags* argument. If `__main__` does not already exist, it is created. Returns `0` on success or `-1` if an exception was raised. If there was an error, there is no way to get the exception information. For the meaning of *flags*, see below.

Note that if an otherwise unhandled `SystemExit` is raised, this function will not return `-1`, but exit the process, as long as `PyConfig.inspect` is zero.

int **PyRun_SimpleFile**(FILE *fp, const char *filename)

This is a simplified interface to `PyRun_SimpleFileExFlags()` below, leaving *closeit* set to `0` and *flags* set to `NULL`.

int **PyRun_SimpleFileEx**(FILE *fp, const char *filename, int closeit)

This is a simplified interface to `PyRun_SimpleFileExFlags()` below, leaving *flags* set to `NULL`.

int **PyRun_SimpleFileExFlags**(FILE *fp, const char *filename, int closeit, *PyCompilerFlags* *flags)

Similar to `PyRun_SimpleStringFlags()`, but the Python source code is read from *fp* instead of an in--memory string. *filename* should be the name of the file, it is decoded from *filesystem encoding and error handler*. If *closeit* is true, the file is closed before `PyRun_SimpleFileExFlags()` returns.

> **Informacja**
>
> On Windows, *fp* should be opened as binary mode (e.g. `fopen(filename, "rb")`). Otherwise, Python may not handle script file with LF line ending correctly.

int **PyRun_InteractiveOne**(FILE *fp, const char *filename)

This is a simplified interface to `PyRun_InteractiveOneFlags()` below, leaving *flags* set to `NULL`.

int **PyRun_InteractiveOneFlags**(FILE *fp, const char *filename, *PyCompilerFlags* *flags)

Read and execute a single statement from a file associated with an interactive device according to the *flags* argument. The user will be prompted using `sys.ps1` and `sys.ps2`. *filename* is decoded from the *filesystem encoding and error handler*.

Returns `0` when the input was executed successfully, `-1` if there was an exception, or an error code from the `errcode.h` include file distributed as part of Python if there was a parse error. (Note that `errcode.h` is not included by `Python.h`, so must be included specifically if needed.)

int **PyRun_InteractiveLoop**(FILE *fp, const char *filename)

This is a simplified interface to `PyRun_InteractiveLoopFlags()` below, leaving *flags* set to `NULL`.

int **PyRun_InteractiveLoopFlags**(FILE *fp, const char *filename, *PyCompilerFlags* *flags)

Read and execute statements from a file associated with an interactive device until EOF is reached. The user will be prompted using `sys.ps1` and `sys.ps2`. *filename* is decoded from the *filesystem encoding and error handler*. Returns `0` at EOF or a negative number upon failure.

int (***PyOS_InputHook**)(void)

Część stabilnego ABI. Can be set to point to a function with the prototype `int func(void)`. The function will be called when Python's interpreter prompt is about to become idle and wait for user input from the terminal. The return value is ignored. Overriding this hook can be used to integrate the interpreter's prompt with other event loops, as done in the `Modules/_tkinter.c` in the Python source code.

Zmienione w wersji 3.12: This function is only called from the *main interpreter*.

char *(***PyOS_ReadlineFunctionPointer**)(FILE*, FILE*, const char*)

Can be set to point to a function with the prototype `char *func(FILE *stdin, FILE *stdout, char *prompt)`, overriding the default function used to read a single line of input at the interpreter's prompt. The function is expected to output the string *prompt* if it's not `NULL`, and then read a line of input from the provided standard input file, returning the resulting string. For example, The `readline` module sets this hook to provide line-editing and tab-completion features.

The result must be a string allocated by `PyMem_RawMalloc()` or `PyMem_RawRealloc()`, or `NULL` if an error occurred.

Zmienione w wersji 3.4: The result must be allocated by `PyMem_RawMalloc()` or `PyMem_RawRealloc()`, instead of being allocated by `PyMem_Malloc()` or `PyMem_Realloc()`.

Zmienione w wersji 3.12: This function is only called from the *main interpreter*.

PyObject ***PyRun_String**(const char *str, int start, *PyObject* *globals, *PyObject* *locals)

Wartość zwracana: nowa referencja. This is a simplified interface to `PyRun_StringFlags()` below, leaving *flags* set to `NULL`.

PyObject ***PyRun_StringFlags**(const char *str, int start, *PyObject* *globals, *PyObject* *locals, *PyCompilerFlags* *flags)

Wartość zwracana: nowa referencja. Execute Python source code from *str* in the context specified by the objects *globals* and *locals* with the compiler flags specified by *flags*. *globals* must be a dictionary; *locals* can be any object that implements the mapping protocol. The parameter *start* specifies the start token that should be used to parse the source code.

Returns the result of executing the code as a Python object, or `NULL` if an exception was raised.

PyObject ***PyRun_File**(FILE *fp, const char *filename, int start, *PyObject* *globals, *PyObject* *locals)

Wartość zwracana: nowa referencja. This is a simplified interface to `PyRun_FileExFlags()` below, leaving *closeit* set to `0` and *flags* set to `NULL`.

PyObject ***PyRun_FileEx**(FILE *fp, const char *filename, int start, *PyObject* *globals, *PyObject* *locals, int closeit)

Wartość zwracana: nowa referencja. This is a simplified interface to `PyRun_FileExFlags()` below, leaving *flags* set to `NULL`.

PyObject ***PyRun_FileFlags**(FILE *fp, const char *filename, int start, *PyObject* *globals, *PyObject* *locals, *PyCompilerFlags* *flags)

Wartość zwracana: nowa referencja. This is a simplified interface to `PyRun_FileExFlags()` below, leaving *closeit* set to `0`.

PyObject ***PyRun_FileExFlags**(FILE *fp, const char *filename, int start, *PyObject* *globals, *PyObject* *locals, int closeit, *PyCompilerFlags* *flags)

Wartość zwracana: nowa referencja. Similar to `PyRun_StringFlags()`, but the Python source code is read from *fp* instead of an in-memory string. *filename* should be the name of the file, it is decoded from the *filesystem encoding and error handler*. If *closeit* is true, the file is closed before `PyRun_FileExFlags()` returns.

PyObject ***Py_CompileString**(const char *str, const char *filename, int start)

Wartość zwracana: nowa referencja. Część stabilnego ABI. This is a simplified interface to `Py_CompileStringFlags()` below, leaving *flags* set to `NULL`.

PyObject ***Py_CompileStringFlags**(const char *str, const char *filename, int start, *PyCompilerFlags* *flags)

Wartość zwracana: nowa referencja. This is a simplified interface to `Py_CompileStringExFlags()` below, with *optimize* set to `-1`.

PyObject ***Py_CompileStringObject**(const char *str, *PyObject* *filename, int start, *PyCompilerFlags* *flags, int optimize)

Wartość zwracana: nowa referencja. Parse and compile the Python source code in *str*, returning the resulting code object. The start token is given by *start*; this can be used to constrain the code which can be compiled and should be `Py_eval_input`, `Py_file_input`, or `Py_single_input`. The filename specified by *filename* is used to construct the code object and may appear in tracebacks or `SyntaxError` exception messages. This returns `NULL` if the code cannot be parsed or compiled.

The integer *optimize* specifies the optimization level of the compiler; a value of `-1` selects the optimization level of the interpreter as given by `-O` options. Explicit levels are `0` (no optimization; `__debug__` is true), `1` (asserts are removed, `__debug__` is false) or `2` (docstrings are removed too).

Added in version 3.4.

PyObject ***Py_CompileStringExFlags**(const char *str, const char *filename, int start, *PyCompilerFlags* *flags, int optimize)

Wartość zwracana: nowa referencja. Like `Py_CompileStringObject()`, but *filename* is a byte string decoded from the *filesystem encoding and error handler*.

Added in version 3.2.

PyObject ***PyEval_EvalCode*** (*PyObject* *co, *PyObject* *globals, *PyObject* *locals)

Wartość zwracana: nowa referencja. Część stabilnego ABI. This is a simplified interface to `PyEval_EvalCodeEx()`, with just the code object, and global and local variables. The other arguments are set to `NULL`.

PyObject ***PyEval_EvalCodeEx*** (*PyObject* *co, *PyObject* *globals, *PyObject* *locals, *PyObject* *const *args, int argcount, *PyObject* *const *kws, int kwcount, *PyObject* *const *defs, int defcount, *PyObject* *kwdefs, *PyObject* *closure)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Evaluate a precompiled code object, given a particular environment for its evaluation. This environment consists of a dictionary of global variables, a mapping object of local variables, arrays of arguments, keywords and defaults, a dictionary of default values for *keyword-only* arguments and a closure tuple of cells.

PyObject ***PyEval_EvalFrame*** (*PyFrameObject* *f)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Evaluate an execution frame. This is a simplified interface to `PyEval_EvalFrameEx()`, for backward compatibility.

PyObject ***PyEval_EvalFrameEx*** (*PyFrameObject* *f, int throwflag)

Wartość zwracana: nowa referencja. Część stabilnego ABI. This is the main, unvarnished function of Python interpretation. The code object associated with the execution frame *f* is executed, interpreting bytecode and executing calls as needed. The additional *throwflag* parameter can mostly be ignored - if true, then it causes an exception to immediately be thrown; this is used for the `throw()` methods of generator objects.

Zmienione w wersji 3.4: This function now includes a debug assertion to help ensure that it does not silently discard an active exception.

int ***PyEval_MergeCompilerFlags*** (*PyCompilerFlags* *cf)

This function changes the flags of the current evaluation frame, and returns true on success, false on failure.

int ***Py_eval_input***

The start symbol from the Python grammar for isolated expressions; for use with `Py_CompileString()`.

int ***Py_file_input***

The start symbol from the Python grammar for sequences of statements as read from a file or other source; for use with `Py_CompileString()`. This is the symbol to use when compiling arbitrarily long Python source code.

int ***Py_single_input***

The start symbol from the Python grammar for a single statement; for use with `Py_CompileString()`. This is the symbol used for the interactive interpreter loop.

struct ***PyCompilerFlags***

This is the structure used to hold compiler flags. In cases where code is only being compiled, it is passed as `int flags`, and in cases where code is being executed, it is passed as `PyCompilerFlags *flags`. In this case, `from __future__ import` can modify *flags*.

Whenever `PyCompilerFlags *flags` is `NULL`, `cf_flags` is treated as equal to `0`, and any modification due to `from __future__ import` is discarded.

int ***cf_flags***

Compiler flags.

int ***cf_feature_version***

cf_feature_version is the minor Python version. It should be initialized to `PY_MINOR_VERSION`.

The field is ignored by default, it is used if and only if `PyCF_ONLY_AST` flag is set in `cf_flags`.

Zmienione w wersji 3.8: Added *cf_feature_version* field.

int ***CO_FUTURE_DIVISION***

This bit can be set in *flags* to cause division operator `/` to be interpreted as „true division" according to **PEP 238**.

ROZDZIAŁ 4

Reference Counting

The functions and macros in this section are used for managing reference counts of Python objects.

Py_ssize_t **Py_REFCNT**(*PyObject* *o)

Get the reference count of the Python object *o*.

Note that the returned value may not actually reflect how many references to the object are actually held. For example, some objects are *immortal* and have a very high refcount that does not reflect the actual number of references. Consequently, do not rely on the returned value to be accurate, other than a value of 0 or 1.

Use the `Py_SET_REFCNT()` function to set an object reference count.

Zmienione w wersji 3.10: `Py_REFCNT()` is changed to the inline static function.

Zmienione w wersji 3.11: The parameter type is no longer `const PyObject*`.

void **Py_SET_REFCNT**(*PyObject* *o, *Py_ssize_t* refcnt)

Set the object *o* reference counter to *refcnt*.

On Python build with Free Threading, if *refcnt* is larger than `UINT32_MAX`, the object is made *immortal*.

This function has no effect on *immortal* objects.

Added in version 3.9.

Zmienione w wersji 3.12: Immortal objects are not modified.

void **Py_INCREF**(*PyObject* *o)

Indicate taking a new *strong reference* to object *o*, indicating it is in use and should not be destroyed.

This function has no effect on *immortal* objects.

This function is usually used to convert a *borrowed reference* to a *strong reference* in-place. The `Py_NewRef()` function can be used to create a new *strong reference*.

When done using the object, release is by calling `Py_DECREF()`.

The object must not be `NULL`; if you aren't sure that it isn't `NULL`, use `Py_XINCREF()`.

Do not expect this function to actually modify *o* in any way. For at least **some objects**, this function has no effect.

Zmienione w wersji 3.12: Immortal objects are not modified.

void **Py_XINCREF**(*PyObject* *o)

Similar to `Py_INCREF()`, but the object *o* can be `NULL`, in which case this has no effect.

See also `Py_XNewRef()`.

PyObject ***Py_NewRef**(*PyObject* *o)

Część stabilnego ABI *od wersji 3.10.* Create a new *strong reference* to an object: call `Py_INCREF()` on *o* and return the object *o*.

When the *strong reference* is no longer needed, `Py_DECREF()` should be called on it to release the reference.

The object *o* must not be `NULL`; use `Py_XNewRef()` if *o* can be `NULL`.

Dla przykładu:

```
Py_INCREF(obj);
self->attr = obj;
```

can be written as:

```
self->attr = Py_NewRef(obj);
```

See also `Py_INCREF()`.

Added in version 3.10.

PyObject ***Py_XNewRef**(*PyObject* *o)

Część stabilnego ABI *od wersji 3.10.* Similar to `Py_NewRef()`, but the object *o* can be NULL.

If the object *o* is `NULL`, the function just returns `NULL`.

Added in version 3.10.

void **Py_DECREF**(*PyObject* *o)

Release a *strong reference* to object *o*, indicating the reference is no longer used.

This function has no effect on *immortal* objects.

Once the last *strong reference* is released (i.e. the object's reference count reaches 0), the object's type's deallocation function (which must not be `NULL`) is invoked.

This function is usually used to delete a *strong reference* before exiting its scope.

The object must not be `NULL`; if you aren't sure that it isn't `NULL`, use `Py_XDECREF()`.

Do not expect this function to actually modify *o* in any way. For at least **some objects**, this function has no effect.

> **Ostrzeżenie**
>
> The deallocation function can cause arbitrary Python code to be invoked (e.g. when a class instance with a `__del__()` method is deallocated). While exceptions in such code are not propagated, the executed code has free access to all Python global variables. This means that any object that is reachable from a global variable should be in a consistent state before `Py_DECREF()` is invoked. For example, code to delete an object from a list should copy a reference to the deleted object in a temporary variable, update the list data structure, and then call `Py_DECREF()` for the temporary variable.

Zmienione w wersji 3.12: Immortal objects are not modified.

void **Py_XDECREF**(*PyObject* *o)

Similar to `Py_DECREF()`, but the object *o* can be `NULL`, in which case this has no effect. The same warning from `Py_DECREF()` applies here as well.

void **Py_CLEAR** (*PyObject* *o)

Release a *strong reference* for object *o*. The object may be `NULL`, in which case the macro has no effect; otherwise the effect is the same as for `Py_DECREF()`, except that the argument is also set to `NULL`. The warning for `Py_DECREF()` does not apply with respect to the object passed because the macro carefully uses a temporary variable and sets the argument to `NULL` before releasing the reference.

It is a good idea to use this macro whenever releasing a reference to an object that might be traversed during garbage collection.

Zmienione w wersji 3.12: The macro argument is now only evaluated once. If the argument has side effects, these are no longer duplicated.

void **Py_IncRef** (*PyObject* *o)

Część stabilnego ABI. Indicate taking a new *strong reference* to object *o*. A function version of `Py_XINCREF()`. It can be used for runtime dynamic embedding of Python.

void **Py_DecRef** (*PyObject* *o)

Część stabilnego ABI. Release a *strong reference* to object *o*. A function version of `Py_XDECREF()`. It can be used for runtime dynamic embedding of Python.

Py_SETREF (dst, src)

Macro safely releasing a *strong reference* to object *dst* and setting *dst* to *src*.

As in case of `Py_CLEAR()`, „the obvious" code can be deadly:

```
Py_DECREF(dst);
dst = src;
```

The safe way is:

```
Py_SETREF(dst, src);
```

That arranges to set *dst* to *src* _before_ releasing the reference to the old value of *dst*, so that any code triggered as a side-effect of *dst* getting torn down no longer believes *dst* points to a valid object.

Added in version 3.6.

Zmienione w wersji 3.12: The macro arguments are now only evaluated once. If an argument has side effects, these are no longer duplicated.

Py_XSETREF (dst, src)

Variant of `Py_SETREF` macro that uses `Py_XDECREF()` instead of `Py_DECREF()`.

Added in version 3.6.

Zmienione w wersji 3.12: The macro arguments are now only evaluated once. If an argument has side effects, these are no longer duplicated.

ROZDZIAŁ 5

Obsługa sytuacji wyjątkowych

The functions described in this chapter will let you handle and raise Python exceptions. It is important to understand some of the basics of Python exception handling. It works somewhat like the POSIX `errno` variable: there is a global indicator (per thread) of the last error that occurred. Most C API functions don't clear this on success, but will set it to indicate the cause of the error on failure. Most C API functions also return an error indicator, usually `NULL` if they are supposed to return a pointer, or `-1` if they return an integer (exception: the `PyArg_*` functions return `1` for success and `0` for failure).

Concretely, the error indicator consists of three object pointers: the exception's type, the exception's value, and the traceback object. Any of those pointers can be `NULL` if non-set (although some combinations are forbidden, for example you can't have a non-`NULL` traceback if the exception type is `NULL`).

Gdy zadanie musi zawieźć z powodu błędu zadania które wywołało, ogólnie nie ustawia ona wskaźnika błędu; podzadanie które zostało wywołane już go ustawiła. Jest on odpowiedzialny albo za obsługę błędu i wyczyszczenie wskaźnika sytuacji wyjątkowej lub powrót po sprzątnięciu jakichkolwiek zasobów które utrzymuje (takich jak odwołania do przedmiotów lub zajęte pamięci); *nie* powinien kontynuować zwyczajnie jeśli nie jest przygotowany do obsługi błędu. Jeśli kończy z powodu błędu, istotne jest zwrócenie uwagi wołającego że został zgłoszony błąd. Jeśli błąd nie jest obsługiwany lub propagowany właściwie, dodatkowe odwołania do sprzęgu języka pytonowskiego/C mogą nie zachowywać się tak, jak planowano i mogą zawieźć w nieoczekiwane sposoby.

Informacja

The error indicator is **not** the result of `sys.exc_info()`. The former corresponds to an exception that is not yet caught (and is therefore still propagating), while the latter returns an exception after it is caught (and has therefore stopped propagating).

5.1 Printing and clearing

void **`PyErr_Clear`**()

Część stabilnego ABI. Clear the error indicator. If the error indicator is not set, there is no effect.

void **`PyErr_PrintEx`**(int set_sys_last_vars)

Część stabilnego ABI. Print a standard traceback to `sys.stderr` and clear the error indicator. **Unless** the error is a `SystemExit`, in that case no traceback is printed and the Python process will exit with the error code specified by the `SystemExit` instance.

Call this function **only** when the error indicator is set. Otherwise it will cause a fatal error!

If *set_sys_last_vars* is nonzero, the variable `sys.last_exc` is set to the printed exception. For backwards compatibility, the deprecated variables `sys.last_type`, `sys.last_value` and `sys.last_traceback` are also set to the type, value and traceback of this exception, respectively.

Zmienione w wersji 3.12: The setting of `sys.last_exc` was added.

void **PyErr_Print**()

Część stabilnego ABI. Alias for `PyErr_PrintEx(1)`.

void **PyErr_WriteUnraisable**(*PyObject* *obj)

Część stabilnego ABI. Call `sys.unraisablehook()` using the current exception and *obj* argument.

This utility function prints a warning message to `sys.stderr` when an exception has been set but it is impossible for the interpreter to actually raise the exception. It is used, for example, when an exception occurs in an `__del__()` method.

The function is called with a single argument *obj* that identifies the context in which the unraisable exception occurred. If possible, the repr of *obj* will be printed in the warning message. If *obj* is `NULL`, only the traceback is printed.

An exception must be set when calling this function.

Zmienione w wersji 3.4: Print a traceback. Print only traceback if *obj* is `NULL`.

Zmienione w wersji 3.8: Use `sys.unraisablehook()`.

void **PyErr_FormatUnraisable**(const char *format, ...)

Similar to `PyErr_WriteUnraisable()`, but the *format* and subsequent parameters help format the warning message; they have the same meaning and values as in `PyUnicode_FromFormat()`. `PyErr_WriteUnraisable(obj)` is roughly equivalent to `PyErr_FormatUnraisable("Exception ignored in: %R", obj)`. If *format* is `NULL`, only the traceback is printed.

Added in version 3.13.

void **PyErr_DisplayException**(*PyObject* *exc)

Część stabilnego ABI *od wersji 3.12*. Print the standard traceback display of `exc` to `sys.stderr`, including chained exceptions and notes.

Added in version 3.12.

5.2 Raising exceptions

These functions help you set the current thread's error indicator. For convenience, some of these functions will always return a `NULL` pointer for use in a `return` statement.

void **PyErr_SetString**(*PyObject* *type, const char *message)

Część stabilnego ABI. This is the most common way to set the error indicator. The first argument specifies the exception type; it is normally one of the standard exceptions, e.g. `PyExc_RuntimeError`. You need not create a new *strong reference* to it (e.g. with `Py_INCREF()`). The second argument is an error message; it is decoded from `'utf-8'`.

void **PyErr_SetObject**(*PyObject* *type, *PyObject* *value)

Część stabilnego ABI. This function is similar to `PyErr_SetString()` but lets you specify an arbitrary Python object for the „value" of the exception.

PyObject ***PyErr_Format**(*PyObject* *exception, const char *format, ...)

Wartość zwracana: zawsze NULL. Część stabilnego ABI. This function sets the error indicator and returns `NULL`. *exception* should be a Python exception class. The *format* and subsequent parameters help format the error message; they have the same meaning and values as in `PyUnicode_FromFormat()`. *format* is an ASCII--encoded string.

PyObject ***PyErr_FormatV*** (*PyObject* *exception, const char *format, va_list vargs)

Wartość zwracana: zawsze NULL. Część stabilnego ABI *od wersji 3.5.* Same as `PyErr_Format()`, but taking a `va_list` argument rather than a variable number of arguments.

Added in version 3.5.

void ***PyErr_SetNone*** (*PyObject* *type)

Część stabilnego ABI. This is a shorthand for `PyErr_SetObject(type, Py_None)`.

int ***PyErr_BadArgument*** ()

Część stabilnego ABI. This is a shorthand for `PyErr_SetString(PyExc_TypeError, message)`, where *message* indicates that a built-in operation was invoked with an illegal argument. It is mostly for internal use.

PyObject ***PyErr_NoMemory*** ()

Wartość zwracana: zawsze NULL. Część stabilnego ABI. This is a shorthand for `PyErr_SetNone(PyExc_MemoryError)`; it returns `NULL` so an object allocation function can write `return PyErr_NoMemory();` when it runs out of memory.

PyObject ***PyErr_SetFromErrno*** (*PyObject* *type)

Wartość zwracana: zawsze NULL. Część stabilnego ABI. This is a convenience function to raise an exception when a C library function has returned an error and set the C variable `errno`. It constructs a tuple object whose first item is the integer `errno` value and whose second item is the corresponding error message (gotten from `strerror()`), and then calls `PyErr_SetObject(type, object)`. On Unix, when the `errno` value is `EINTR`, indicating an interrupted system call, this calls `PyErr_CheckSignals()`, and if that set the error indicator, leaves it set to that. The function always returns `NULL`, so a wrapper function around a system call can write `return PyErr_SetFromErrno(type);` when the system call returns an error.

PyObject ***PyErr_SetFromErrnoWithFilenameObject*** (*PyObject* *type, *PyObject* *filenameObject)

Wartość zwracana: zawsze NULL. Część stabilnego ABI. Similar to `PyErr_SetFromErrno()`, with the additional behavior that if *filenameObject* is not `NULL`, it is passed to the constructor of *type* as a third parameter. In the case of `OSError` exception, this is used to define the `filename` attribute of the exception instance.

PyObject ***PyErr_SetFromErrnoWithFilenameObjects*** (*PyObject* *type, *PyObject* *filenameObject, *PyObject* *filenameObject2)

Wartość zwracana: zawsze NULL. Część stabilnego ABI *od wersji 3.7.* Similar to `PyErr_SetFromErrnoWithFilenameObject()`, but takes a second filename object, for raising errors when a function that takes two filenames fails.

Added in version 3.4.

PyObject ***PyErr_SetFromErrnoWithFilename*** (*PyObject* *type, const char *filename)

Wartość zwracana: zawsze NULL. Część stabilnego ABI. Similar to `PyErr_SetFromErrnoWithFilenameObject()`, but the filename is given as a C string. *filename* is decoded from the *filesystem encoding and error handler*.

PyObject ***PyErr_SetFromWindowsErr*** (int ierr)

Wartość zwracana: zawsze NULL. Część stabilnego ABI *on Windows od wersji 3.7.* This is a convenience function to raise `OSError`. If called with *ierr* of `0`, the error code returned by a call to `GetLastError()` is used instead. It calls the Win32 function `FormatMessage()` to retrieve the Windows description of error code given by *ierr* or `GetLastError()`, then it constructs a `OSError` object with the `winerror` attribute set to the error code, the `strerror` attribute set to the corresponding error message (gotten from `FormatMessage()`), and then calls `PyErr_SetObject(PyExc_OSError, object)`. This function always returns `NULL`.

Availability: Windows.

PyObject ***PyErr_SetExcFromWindowsErr*** (*PyObject* *type, int ierr)

Wartość zwracana: zawsze NULL. Część stabilnego ABI *on Windows od wersji 3.7.* Similar to `PyErr_SetFromWindowsErr()`, with an additional parameter specifying the exception type to be raised.

Availability: Windows.

PyObject ***PyErr_SetFromWindowsErrWithFilename**(int ierr, const char *filename)

Wartość zwracana: zawsze NULL. Część stabilnego ABI *on Windows od wersji 3.7.* Similar to `PyErr_SetFromWindowsErr()`, with the additional behavior that if *filename* is not `NULL`, it is decoded from the filesystem encoding (`os.fsdecode()`) and passed to the constructor of `OSError` as a third parameter to be used to define the `filename` attribute of the exception instance.

Availability: Windows.

PyObject ***PyErr_SetExcFromWindowsErrWithFilenameObject**(*PyObject* *type, int ierr, *PyObject* *filename)

Wartość zwracana: zawsze NULL. Część stabilnego ABI *on Windows od wersji 3.7.* Similar to `PyErr_SetExcFromWindowsErr()`, with the additional behavior that if *filename* is not `NULL`, it is passed to the constructor of `OSError` as a third parameter to be used to define the `filename` attribute of the exception instance.

Availability: Windows.

PyObject ***PyErr_SetExcFromWindowsErrWithFilenameObjects**(*PyObject* *type, int ierr, *PyObject* *filename, *PyObject* *filename2)

Wartość zwracana: zawsze NULL. Część stabilnego ABI *on Windows od wersji 3.7.* Similar to `PyErr_SetExcFromWindowsErrWithFilenameObject()`, but accepts a second filename object.

Availability: Windows.

Added in version 3.4.

PyObject ***PyErr_SetExcFromWindowsErrWithFilename**(*PyObject* *type, int ierr, const char *filename)

Wartość zwracana: zawsze NULL. Część stabilnego ABI *on Windows od wersji 3.7.* Similar to `PyErr_SetFromWindowsErrWithFilename()`, with an additional parameter specifying the exception type to be raised.

Availability: Windows.

PyObject ***PyErr_SetImportError**(*PyObject* *msg, *PyObject* *name, *PyObject* *path)

Wartość zwracana: zawsze NULL. Część stabilnego ABI *od wersji 3.7.* This is a convenience function to raise `ImportError`. *msg* will be set as the exception's message string. *name* and *path*, both of which can be `NULL`, will be set as the `ImportError`'s respective `name` and `path` attributes.

Added in version 3.3.

PyObject ***PyErr_SetImportErrorSubclass**(*PyObject* *exception, *PyObject* *msg, *PyObject* *name, *PyObject* *path)

Wartość zwracana: zawsze NULL. Część stabilnego ABI *od wersji 3.6.* Much like `PyErr_SetImportError()` but this function allows for specifying a subclass of `ImportError` to raise.

Added in version 3.6.

void **PyErr_SyntaxLocationObject**(*PyObject* *filename, int lineno, int col_offset)

Set file, line, and offset information for the current exception. If the current exception is not a `SyntaxError`, then it sets additional attributes, which make the exception printing subsystem think the exception is a `SyntaxError`.

Added in version 3.4.

void **PyErr_SyntaxLocationEx**(const char *filename, int lineno, int col_offset)

Część stabilnego ABI *od wersji 3.7.* Like `PyErr_SyntaxLocationObject()`, but *filename* is a byte string decoded from the *filesystem encoding and error handler*.

Added in version 3.2.

void **PyErr_SyntaxLocation**(const char *filename, int lineno)

Część stabilnego ABI. Like `PyErr_SyntaxLocationEx()`, but the *col_offset* parameter is omitted.

void **PyErr_BadInternalCall**()

Część stabilnego ABI. This is a shorthand for `PyErr_SetString(PyExc_SystemError, message)`, where *message* indicates that an internal operation (e.g. a Python/C API function) was invoked with an illegal argument. It is mostly for internal use.

5.3 Issuing warnings

Use these functions to issue warnings from C code. They mirror similar functions exported by the Python `warnings` module. They normally print a warning message to *sys.stderr*; however, it is also possible that the user has specified that warnings are to be turned into errors, and in that case they will raise an exception. It is also possible that the functions raise an exception because of a problem with the warning machinery. The return value is `0` if no exception is raised, or `-1` if an exception is raised. (It is not possible to determine whether a warning message is actually printed, nor what the reason is for the exception; this is intentional.) If an exception is raised, the caller should do its normal exception handling (for example, `Py_DECREF()` owned references and return an error value).

int **PyErr_WarnEx**(*PyObject* *category, const char *message, *Py_ssize_t* stack_level)

Część stabilnego ABI. Issue a warning message. The *category* argument is a warning category (see below) or `NULL`; the *message* argument is a UTF-8 encoded string. *stack_level* is a positive number giving a number of stack frames; the warning will be issued from the currently executing line of code in that stack frame. A *stack_level* of 1 is the function calling `PyErr_WarnEx()`, 2 is the function above that, and so forth.

Warning categories must be subclasses of `PyExc_Warning`; `PyExc_Warning` is a subclass of `PyExc_Exception`; the default warning category is `PyExc_RuntimeWarning`. The standard Python warning categories are available as global variables whose names are enumerated at *Standard Warning Categories*.

For information about warning control, see the documentation for the `warnings` module and the `-W` option in the command line documentation. There is no C API for warning control.

int **PyErr_WarnExplicitObject**(*PyObject* *category, *PyObject* *message, *PyObject* *filename, int lineno, *PyObject* *module, *PyObject* *registry)

Issue a warning message with explicit control over all warning attributes. This is a straightforward wrapper around the Python function `warnings.warn_explicit()`; see there for more information. The *module* and *registry* arguments may be set to `NULL` to get the default effect described there.

Added in version 3.4.

int **PyErr_WarnExplicit**(*PyObject* *category, const char *message, const char *filename, int lineno, const char *module, *PyObject* *registry)

Część stabilnego ABI. Similar to `PyErr_WarnExplicitObject()` except that *message* and *module* are UTF--8 encoded strings, and *filename* is decoded from the *filesystem encoding and error handler*.

int **PyErr_WarnFormat**(*PyObject* *category, *Py_ssize_t* stack_level, const char *format, ...)

Część stabilnego ABI. Function similar to `PyErr_WarnEx()`, but use `PyUnicode_FromFormat()` to format the warning message. *format* is an ASCII-encoded string.

Added in version 3.2.

int **PyErr_ResourceWarning**(*PyObject* *source, *Py_ssize_t* stack_level, const char *format, ...)

Część stabilnego ABI *od wersji 3.6*. Function similar to `PyErr_WarnFormat()`, but *category* is `ResourceWarning` and it passes *source* to `warnings.WarningMessage`.

Added in version 3.6.

5.4 Querying the error indicator

PyObject ***PyErr_Occurred**()

Wartość zwracana: pożyczona referencja. Część stabilnego ABI. Test whether the error indicator is set. If set, return the exception *type* (the first argument to the last call to one of the `PyErr_Set*` functions or to `PyErr_Restore()`). If not set, return `NULL`. You do not own a reference to the return value, so you do not need to `Py_DECREF()` it.

The caller must hold the GIL.

> **Informacja**
>
> Do not compare the return value to a specific exception; use `PyErr_ExceptionMatches()` instead, shown below. (The comparison could easily fail since the exception may be an instance instead of a class, in the case of a class exception, or it may be a subclass of the expected exception.)

int **PyErr_ExceptionMatches**(*PyObject* *exc)

Część stabilnego ABI. Equivalent to `PyErr_GivenExceptionMatches(PyErr_Occurred(), exc)`. This should only be called when an exception is actually set; a memory access violation will occur if no exception has been raised.

int **PyErr_GivenExceptionMatches**(*PyObject* *given, *PyObject* *exc)

Część stabilnego ABI. Return true if the *given* exception matches the exception type in *exc*. If *exc* is a class object, this also returns true when *given* is an instance of a subclass. If *exc* is a tuple, all exception types in the tuple (and recursively in subtuples) are searched for a match.

PyObject ***PyErr_GetRaisedException**(void)

Wartość zwracana: nowa referencja. Część stabilnego ABI *od wersji 3.12.* Return the exception currently being raised, clearing the error indicator at the same time. Return `NULL` if the error indicator is not set.

This function is used by code that needs to catch exceptions, or code that needs to save and restore the error indicator temporarily.

Na przykład:

```
{
   PyObject *exc = PyErr_GetRaisedException();

   /* ... code that might produce other errors ... */

   PyErr_SetRaisedException(exc);
}
```

> **Zobacz także**
>
> `PyErr_GetHandledException()`, to save the exception currently being handled.

Added in version 3.12.

void **PyErr_SetRaisedException**(*PyObject* *exc)

Część stabilnego ABI *od wersji 3.12.* Set *exc* as the exception currently being raised, clearing the existing exception if one is set.

> **Ostrzeżenie**
>
> This call steals a reference to *exc*, which must be a valid exception.

Added in version 3.12.

void **PyErr_Fetch**(*PyObject* **ptype, *PyObject* **pvalue, *PyObject* **ptraceback)

Część stabilnego ABI. Niezalecane od wersji 3.12: Use `PyErr_GetRaisedException()` instead.

Retrieve the error indicator into three variables whose addresses are passed. If the error indicator is not set, set all three variables to `NULL`. If it is set, it will be cleared and you own a reference to each object retrieved. The value and traceback object may be `NULL` even when the type object is not.

Informacja

This function is normally only used by legacy code that needs to catch exceptions or save and restore the error indicator temporarily.

Na przykład:

```
{
   PyObject *type, *value, *traceback;
   PyErr_Fetch(&type, &value, &traceback);

   /* ... code that might produce other errors ... */

   PyErr_Restore(type, value, traceback);
}
```

void **PyErr_Restore**(*PyObject* *type, *PyObject* *value, *PyObject* *traceback)

Część stabilnego ABI. Niezalecane od wersji 3.12: Use `PyErr_SetRaisedException()` instead.

Set the error indicator from the three objects, *type*, *value*, and *traceback*, clearing the existing exception if one is set. If the objects are `NULL`, the error indicator is cleared. Do not pass a `NULL` type and non-`NULL` value or traceback. The exception type should be a class. Do not pass an invalid exception type or value. (Violating these rules will cause subtle problems later.) This call takes away a reference to each object: you must own a reference to each object before the call and after the call you no longer own these references. (If you don't understand this, don't use this function. I warned you.)

Informacja

This function is normally only used by legacy code that needs to save and restore the error indicator temporarily. Use `PyErr_Fetch()` to save the current error indicator.

void **PyErr_NormalizeException**(*PyObject* **exc, *PyObject* **val, *PyObject* **tb)

Część stabilnego ABI. Niezalecane od wersji 3.12: Use `PyErr_GetRaisedException()` instead, to avoid any possible de-normalization.

Under certain circumstances, the values returned by `PyErr_Fetch()` below can be „unnormalized", meaning that `*exc` is a class object but `*val` is not an instance of the same class. This function can be used to instantiate the class in that case. If the values are already normalized, nothing happens. The delayed normalization is implemented to improve performance.

Informacja

This function *does not* implicitly set the `__traceback__` attribute on the exception value. If setting the traceback appropriately is desired, the following additional snippet is needed:

```
if (tb != NULL) {
  PyException_SetTraceback(val, tb);
}
```

PyObject ***PyErr_GetHandledException**(void)

Część stabilnego ABI *od wersji 3.11.* Retrieve the active exception instance, as would be returned by `sys.exception()`. This refers to an exception that was *already caught*, not to an exception that was freshly raised. Returns a new reference to the exception or `NULL`. Does not modify the interpreter's exception state.

> **Informacja**
>
> This function is not normally used by code that wants to handle exceptions. Rather, it can be used when code needs to save and restore the exception state temporarily. Use `PyErr_SetHandledException()` to restore or clear the exception state.

Added in version 3.11.

void **`PyErr_SetHandledException`**(*PyObject* *exc)

Część stabilnego ABI *od wersji 3.11.* Set the active exception, as known from `sys.exception()`. This refers to an exception that was *already caught*, not to an exception that was freshly raised. To clear the exception state, pass `NULL`.

> **Informacja**
>
> This function is not normally used by code that wants to handle exceptions. Rather, it can be used when code needs to save and restore the exception state temporarily. Use `PyErr_GetHandledException()` to get the exception state.

Added in version 3.11.

void **`PyErr_GetExcInfo`**(*PyObject* **ptype, *PyObject* **pvalue, *PyObject* **ptraceback)

Część stabilnego ABI *od wersji 3.7.* Retrieve the old-style representation of the exception info, as known from `sys.exc_info()`. This refers to an exception that was *already caught*, not to an exception that was freshly raised. Returns new references for the three objects, any of which may be `NULL`. Does not modify the exception info state. This function is kept for backwards compatibility. Prefer using `PyErr_GetHandledException()`.

> **Informacja**
>
> This function is not normally used by code that wants to handle exceptions. Rather, it can be used when code needs to save and restore the exception state temporarily. Use `PyErr_SetExcInfo()` to restore or clear the exception state.

Added in version 3.3.

void **`PyErr_SetExcInfo`**(*PyObject* *type, *PyObject* *value, *PyObject* *traceback)

Część stabilnego ABI *od wersji 3.7.* Set the exception info, as known from `sys.exc_info()`. This refers to an exception that was *already caught*, not to an exception that was freshly raised. This function steals the references of the arguments. To clear the exception state, pass `NULL` for all three arguments. This function is kept for backwards compatibility. Prefer using `PyErr_SetHandledException()`.

> **Informacja**
>
> This function is not normally used by code that wants to handle exceptions. Rather, it can be used when code needs to save and restore the exception state temporarily. Use `PyErr_GetExcInfo()` to read the exception state.

Added in version 3.3.

Zmienione w wersji 3.11: The `type` and `traceback` arguments are no longer used and can be NULL. The interpreter now derives them from the exception instance (the `value` argument). The function still steals references of all three arguments.

5.5 Signal Handling

int **PyErr_CheckSignals**()

Część stabilnego ABI. This function interacts with Python's signal handling.

If the function is called from the main thread and under the main Python interpreter, it checks whether a signal has been sent to the processes and if so, invokes the corresponding signal handler. If the `signal` module is supported, this can invoke a signal handler written in Python.

The function attempts to handle all pending signals, and then returns `0`. However, if a Python signal handler raises an exception, the error indicator is set and the function returns `-1` immediately (such that other pending signals may not have been handled yet: they will be on the next `PyErr_CheckSignals()` invocation).

If the function is called from a non-main thread, or under a non-main Python interpreter, it does nothing and returns `0`.

This function can be called by long-running C code that wants to be interruptible by user requests (such as by pressing Ctrl-C).

> **Informacja**
>
> The default Python signal handler for `SIGINT` raises the `KeyboardInterrupt` exception.

void **PyErr_SetInterrupt**()

Część stabilnego ABI. Simulate the effect of a `SIGINT` signal arriving. This is equivalent to `PyErr_SetInterruptEx(SIGINT)`.

> **Informacja**
>
> This function is async-signal-safe. It can be called without the *GIL* and from a C signal handler.

int **PyErr_SetInterruptEx**(int signum)

Część stabilnego ABI *od wersji 3.10.* Simulate the effect of a signal arriving. The next time `PyErr_CheckSignals()` is called, the Python signal handler for the given signal number will be called.

This function can be called by C code that sets up its own signal handling and wants Python signal handlers to be invoked as expected when an interruption is requested (for example when the user presses Ctrl-C to interrupt an operation).

If the given signal isn't handled by Python (it was set to `signal.SIG_DFL` or `signal.SIG_IGN`), it will be ignored.

If *signum* is outside of the allowed range of signal numbers, `-1` is returned. Otherwise, `0` is returned. The error indicator is never changed by this function.

> **Informacja**
>
> This function is async-signal-safe. It can be called without the *GIL* and from a C signal handler.

Added in version 3.10.

int **PySignal_SetWakeupFd**(int fd)

This utility function specifies a file descriptor to which the signal number is written as a single byte whenever a signal is received. *fd* must be non-blocking. It returns the previous such file descriptor.

The value `-1` disables the feature; this is the initial state. This is equivalent to `signal.set_wakeup_fd()` in Python, but without any error checking. *fd* should be a valid file descriptor. The function should only be called from the main thread.

Zmienione w wersji 3.5: On Windows, the function now also supports socket handles.

5.6 Exception Classes

PyObject ***PyErr_NewException**(const char *name, *PyObject* *base, *PyObject* *dict)

Wartość zwracana: nowa referencja. Część stabilnego ABI. This utility function creates and returns a new exception class. The *name* argument must be the name of the new exception, a C string of the form `module.classname`. The *base* and *dict* arguments are normally `NULL`. This creates a class object derived from `Exception` (accessible in C as `PyExc_Exception`).

The `__module__` attribute of the new class is set to the first part (up to the last dot) of the *name* argument, and the class name is set to the last part (after the last dot). The *base* argument can be used to specify alternate base classes; it can either be only one class or a tuple of classes. The *dict* argument can be used to specify a dictionary of class variables and methods.

PyObject ***PyErr_NewExceptionWithDoc**(const char *name, const char *doc, *PyObject* *base, *PyObject* *dict)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Same as `PyErr_NewException()`, except that the new exception class can easily be given a docstring: If *doc* is non-`NULL`, it will be used as the docstring for the exception class.

Added in version 3.2.

5.7 Przedmioty Sytuacji Wyjątkowych

PyObject ***PyException_GetTraceback**(*PyObject* *ex)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return the traceback associated with the exception as a new reference, as accessible from Python through the `__traceback__` attribute. If there is no traceback associated, this returns `NULL`.

int **PyException_SetTraceback**(*PyObject* *ex, *PyObject* *tb)

Część stabilnego ABI. Set the traceback associated with the exception to *tb*. Use `Py_None` to clear it.

PyObject ***PyException_GetContext**(*PyObject* *ex)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return the context (another exception instance during whose handling *ex* was raised) associated with the exception as a new reference, as accessible from Python through the `__context__` attribute. If there is no context associated, this returns `NULL`.

void **PyException_SetContext**(*PyObject* *ex, *PyObject* *ctx)

Część stabilnego ABI. Set the context associated with the exception to *ctx*. Use `NULL` to clear it. There is no type check to make sure that *ctx* is an exception instance. This steals a reference to *ctx*.

PyObject ***PyException_GetCause**(*PyObject* *ex)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return the cause (either an exception instance, or `None`, set by `raise ... from ...`) associated with the exception as a new reference, as accessible from Python through the `__cause__` attribute.

void **PyException_SetCause**(*PyObject* *ex, *PyObject* *cause)

Część stabilnego ABI. Set the cause associated with the exception to *cause*. Use `NULL` to clear it. There is no type check to make sure that *cause* is either an exception instance or `None`. This steals a reference to *cause*.

The `__suppress_context__` attribute is implicitly set to `True` by this function.

PyObject ***PyException_GetArgs**(*PyObject* *ex)

Wartość zwracana: nowa referencja. Część stabilnego ABI *od wersji 3.12*. Return `args` of exception *ex*.

void **PyException_SetArgs**(*PyObject* *ex, *PyObject* *args)

Część stabilnego ABI *od wersji 3.12*. Set `args` of exception *ex* to *args*.

PyObject ***PyUnstable_Exc_PrepReraiseStar*** (*PyObject* *orig, *PyObject* *excs)

> To jest *niestabilne API*. Może ulec zmianie bez ostrzeżenia w wydaniach minor.

Implement part of the interpreter's implementation of `except*`. *orig* is the original exception that was caught, and *excs* is the list of the exceptions that need to be raised. This list contains the unhandled part of *orig*, if any, as well as the exceptions that were raised from the `except*` clauses (so they have a different traceback from *orig*) and those that were reraised (and have the same traceback as *orig*). Return the `ExceptionGroup` that needs to be reraised in the end, or `None` if there is nothing to reraise.

Added in version 3.12.

5.8 Unicode Exception Objects

The following functions are used to create and modify Unicode exceptions from C.

PyObject ***PyUnicodeDecodeError_Create*** (const char *encoding, const char *object, *Py_ssize_t* length, *Py_ssize_t* start, *Py_ssize_t* end, const char *reason)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Create a `UnicodeDecodeError` object with the attributes *encoding*, *object*, *length*, *start*, *end* and *reason*. *encoding* and *reason* are UTF-8 encoded strings.

PyObject ***PyUnicodeDecodeError_GetEncoding*** (*PyObject* *exc)

PyObject ***PyUnicodeEncodeError_GetEncoding*** (*PyObject* *exc)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return the *encoding* attribute of the given exception object.

PyObject ***PyUnicodeDecodeError_GetObject*** (*PyObject* *exc)

PyObject ***PyUnicodeEncodeError_GetObject*** (*PyObject* *exc)

PyObject ***PyUnicodeTranslateError_GetObject*** (*PyObject* *exc)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return the *object* attribute of the given exception object.

int ***PyUnicodeDecodeError_GetStart*** (*PyObject* *exc, *Py_ssize_t* *start)

int ***PyUnicodeEncodeError_GetStart*** (*PyObject* *exc, *Py_ssize_t* *start)

int ***PyUnicodeTranslateError_GetStart*** (*PyObject* *exc, *Py_ssize_t* *start)

Część stabilnego ABI. Get the *start* attribute of the given exception object and place it into **start*. *start* must not be `NULL`. Return `0` on success, `-1` on failure.

int ***PyUnicodeDecodeError_SetStart*** (*PyObject* *exc, *Py_ssize_t* start)

int ***PyUnicodeEncodeError_SetStart*** (*PyObject* *exc, *Py_ssize_t* start)

int ***PyUnicodeTranslateError_SetStart*** (*PyObject* *exc, *Py_ssize_t* start)

Część stabilnego ABI. Set the *start* attribute of the given exception object to *start*. Return `0` on success, `-1` on failure.

int ***PyUnicodeDecodeError_GetEnd*** (*PyObject* *exc, *Py_ssize_t* *end)

int ***PyUnicodeEncodeError_GetEnd*** (*PyObject* *exc, *Py_ssize_t* *end)

int ***PyUnicodeTranslateError_GetEnd*** (*PyObject* *exc, *Py_ssize_t* *end)

Część stabilnego ABI. Get the *end* attribute of the given exception object and place it into **end*. *end* must not be `NULL`. Return `0` on success, `-1` on failure.

int ***PyUnicodeDecodeError_SetEnd*** (*PyObject* *exc, *Py_ssize_t* end)

int ***PyUnicodeEncodeError_SetEnd*** (*PyObject* *exc, *Py_ssize_t* end)

int **PyUnicodeTranslateError_SetEnd**(*PyObject* *exc, *Py_ssize_t* end)

Część stabilnego ABI. Set the *end* attribute of the given exception object to *end*. Return `0` on success, `-1` on failure.

PyObject ***PyUnicodeDecodeError_GetReason**(*PyObject* *exc)

PyObject ***PyUnicodeEncodeError_GetReason**(*PyObject* *exc)

PyObject ***PyUnicodeTranslateError_GetReason**(*PyObject* *exc)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return the *reason* attribute of the given exception object.

int **PyUnicodeDecodeError_SetReason**(*PyObject* *exc, const char *reason)

int **PyUnicodeEncodeError_SetReason**(*PyObject* *exc, const char *reason)

int **PyUnicodeTranslateError_SetReason**(*PyObject* *exc, const char *reason)

Część stabilnego ABI. Set the *reason* attribute of the given exception object to *reason*. Return `0` on success, `-1` on failure.

5.9 Kontrola Rekursji

These two functions provide a way to perform safe recursive calls at the C level, both in the core and in extension modules. They are needed if the recursive code does not necessarily invoke Python code (which tracks its recursion depth automatically). They are also not needed for *tp_call* implementations because the *call protocol* takes care of recursion handling.

int **Py_EnterRecursiveCall**(const char *where)

Część stabilnego ABI *od wersji 3.9.* Marks a point where a recursive C-level call is about to be performed.

If `USE_STACKCHECK` is defined, this function checks if the OS stack overflowed using `PyOS_CheckStack()`. If this is the case, it sets a `MemoryError` and returns a nonzero value.

The function then checks if the recursion limit is reached. If this is the case, a `RecursionError` is set and a nonzero value is returned. Otherwise, zero is returned.

where should be a UTF-8 encoded string such as `" in instance check"` to be concatenated to the `RecursionError` message caused by the recursion depth limit.

Zmienione w wersji 3.9: This function is now also available in the *limited API*.

void **Py_LeaveRecursiveCall**(void)

Część stabilnego ABI *od wersji 3.9.* Ends a `Py_EnterRecursiveCall()`. Must be called once for each *successful* invocation of `Py_EnterRecursiveCall()`.

Zmienione w wersji 3.9: This function is now also available in the *limited API*.

Properly implementing `tp_repr` for container types requires special recursion handling. In addition to protecting the stack, `tp_repr` also needs to track objects to prevent cycles. The following two functions facilitate this functionality. Effectively, these are the C equivalent to `reprlib.recursive_repr()`.

int **Py_ReprEnter**(*PyObject* *object)

Część stabilnego ABI. Called at the beginning of the `tp_repr` implementation to detect cycles.

If the object has already been processed, the function returns a positive integer. In that case the `tp_repr` implementation should return a string object indicating a cycle. As examples, `dict` objects return `{...}` and `list` objects return `[...]`.

The function will return a negative integer if the recursion limit is reached. In that case the `tp_repr` implementation should typically return `NULL`.

Otherwise, the function returns zero and the `tp_repr` implementation can continue normally.

void **Py_ReprLeave**(*PyObject* *object)

Część stabilnego ABI. Ends a `Py_ReprEnter()`. Must be called once for each invocation of `Py_ReprEnter()` that returns zero.

5.10 Sztandarowe Sytuacje Wyjątkowe

All standard Python exceptions are available as global variables whose names are `PyExc_` followed by the Python exception name. These have the type `PyObject*`; they are all class objects. For completeness, here are all the variables:

Nazwa C	Nazwa w języku pytonowskim	Notatki
PyExc_BaseException	BaseException	1
PyExc_Exception	Exception	Strona 64, 1
PyExc_ArithmeticError	ArithmeticError	Strona 64, 1
PyExc_AssertionError	AssertionError	
PyExc_AttributeError	AttributeError	
PyExc_BlockingIOError	BlockingIOError	
PyExc_BrokenPipeError	BrokenPipeError	
PyExc_BufferError	BufferError	
PyExc_ChildProcessError	ChildProcessError	
PyExc_ConnectionAbortedErro	ConnectionAbortedError	
PyExc_ConnectionError	ConnectionError	
PyExc_ConnectionRefusedErro	ConnectionRefusedError	
PyExc_ConnectionResetError	ConnectionResetError	
PyExc_EOFError	EOFError	
PyExc_FileExistsError	FileExistsError	
PyExc_FileNotFoundError	FileNotFoundError	
PyExc_FloatingPointError	FloatingPointError	
PyExc_GeneratorExit	GeneratorExit	
PyExc_ImportError	ImportError	
PyExc_IndentationError	IndentationError	
PyExc_IndexError	IndexError	
PyExc_InterruptedError	InterruptedError	
PyExc_IsADirectoryError	IsADirectoryError	
PyExc_KeyError	KeyError	
PyExc_KeyboardInterrupt	KeyboardInterrupt	
PyExc_LookupError	LookupError	Strona 64, 1
PyExc_MemoryError	MemoryError	
PyExc_ModuleNotFoundError	ModuleNotFoundError	
PyExc_NameError	NameError	
PyExc_NotADirectoryError	NotADirectoryError	
PyExc_NotImplementedError	NotImplementedError	
PyExc_OSError	OSError	Strona 64, 1
PyExc_OverflowError	OverflowError	
PyExc_PermissionError	PermissionError	
PyExc_ProcessLookupError	ProcessLookupError	
PyExc_PythonFinalizationErr	PythonFinalizationError	
PyExc_RecursionError	RecursionError	
PyExc_ReferenceError	ReferenceError	
PyExc_RuntimeError	RuntimeError	
PyExc_StopAsyncIteration	StopAsyncIteration	
PyExc_StopIteration	StopIteration	
PyExc_SyntaxError	SyntaxError	
PyExc_SystemError	SystemError	
PyExc_SystemExit	SystemExit	
PyExc_TabError	TabError	
PyExc_TimeoutError	TimeoutError	
PyExc_TypeError	TypeError	
PyExc_UnboundLocalError	UnboundLocalError	
PyExc_UnicodeDecodeError	UnicodeDecodeError	

ciąg dalszy na następnej stronie

Tabela 1 – kontynuacja poprzedniej strony

Nazwa C	Nazwa w języku pytonowskim	Notatki
PyExc_UnicodeEncodeError	UnicodeEncodeError	
PyExc_UnicodeError	UnicodeError	
PyExc_UnicodeTranslateError	UnicodeTranslateError	
PyExc_ValueError	ValueError	
PyExc_ZeroDivisionError	ZeroDivisionError	

Added in version 3.3: PyExc_BlockingIOError, PyExc_BrokenPipeError, PyExc_ChildProcessError, PyExc_ConnectionError, PyExc_ConnectionAbortedError, PyExc_ConnectionRefusedError, PyExc_ConnectionResetError, PyExc_FileExistsError, PyExc_FileNotFoundError, PyExc_InterruptedError, PyExc_IsADirectoryError, PyExc_NotADirectoryError, PyExc_PermissionError, PyExc_ProcessLookupError and PyExc_TimeoutError were introduced following **PEP 3151**.

Added in version 3.5: PyExc_StopAsyncIteration and PyExc_RecursionError.

Added in version 3.6: PyExc_ModuleNotFoundError.

These are compatibility aliases to PyExc_OSError:

Nazwa C	Notatki
PyExc_EnvironmentError	
PyExc_IOError	
PyExc_WindowsError	[2]

Zmienione w wersji 3.3: These aliases used to be separate exception types.

Uwagi:

5.11 Standard Warning Categories

All standard Python warning categories are available as global variables whose names are PyExc_ followed by the Python exception name. These have the type *PyObject**; they are all class objects. For completeness, here are all the variables:

Nazwa C	Nazwa w języku pytonowskim	Notatki
PyExc_Warning	Warning	[3]
PyExc_BytesWarning	BytesWarning	
PyExc_DeprecationWarning	DeprecationWarning	
PyExc_FutureWarning	FutureWarning	
PyExc_ImportWarning	ImportWarning	
PyExc_PendingDeprecationWarning	PendingDeprecationWarning	
PyExc_ResourceWarning	ResourceWarning	
PyExc_RuntimeWarning	RuntimeWarning	
PyExc_SyntaxWarning	SyntaxWarning	
PyExc_UnicodeWarning	UnicodeWarning	
PyExc_UserWarning	UserWarning	

Added in version 3.2: PyExc_ResourceWarning.

Uwagi:

[1] To jest podstawowy rodzaj przedmiotu dla innych sztandarowych sytuacji wyjątkowych.

[2] Zdefiniowane tylko w systemie Windows; Kod chroniony który używa tego przez sprawdzenie czy makrodefinicja preprocesora MS_WINDOWS jest określona.

[3] This is a base class for other standard warning categories.

ROZDZIAŁ 6

Utilities

The functions in this chapter perform various utility tasks, ranging from helping C code be more portable across platforms, using Python modules from C, and parsing function arguments and constructing Python values from C values.

6.1 Operating System Utilities

PyObject ***PyOS_FSPath**(*PyObject* *path)

Wartość zwracana: nowa referencja. Część stabilnego ABI *od wersji 3.6.* Return the file system representation for *path*. If the object is a `str` or `bytes` object, then a new *strong reference* is returned. If the object implements the `os.PathLike` interface, then `__fspath__()` is returned as long as it is a `str` or `bytes` object. Otherwise `TypeError` is raised and `NULL` is returned.

Added in version 3.6.

int **Py_FdIsInteractive**(FILE *fp, const char *filename)

Return true (nonzero) if the standard I/O file *fp* with name *filename* is deemed interactive. This is the case for files for which `isatty(fileno(fp))` is true. If the `PyConfig.interactive` is non-zero, this function also returns true if the *filename* pointer is `NULL` or if the name is equal to one of the strings `'<stdin>'` or `'???'`.

This function must not be called before Python is initialized.

void **PyOS_BeforeFork**()

Część stabilnego ABI *on platforms with fork() od wersji 3.7.* Function to prepare some internal state before a process fork. This should be called before calling `fork()` or any similar function that clones the current process. Only available on systems where `fork()` is defined.

> **Ostrzeżenie**
>
> The C `fork()` call should only be made from the *„main" thread* (of the *„main" interpreter*). The same is true for `PyOS_BeforeFork()`.

Added in version 3.7.

void **PyOS_AfterFork_Parent**()

Część stabilnego ABI *on platforms with fork() od wersji 3.7.* Function to update some internal state after a process fork. This should be called from the parent process after calling `fork()` or any similar function that clones the current process, regardless of whether process cloning was successful. Only available on systems where `fork()` is defined.

> **Ostrzeżenie**
>
> The C `fork()` call should only be made from the *„main" thread* (of the *„main" interpreter*). The same is true for `PyOS_AfterFork_Parent()`.

Added in version 3.7.

void **PyOS_AfterFork_Child**()

Część stabilnego ABI *on platforms with fork() od wersji 3.7.* Function to update internal interpreter state after a process fork. This must be called from the child process after calling `fork()`, or any similar function that clones the current process, if there is any chance the process will call back into the Python interpreter. Only available on systems where `fork()` is defined.

> **Ostrzeżenie**
>
> The C `fork()` call should only be made from the *„main" thread* (of the *„main" interpreter*). The same is true for `PyOS_AfterFork_Child()`.

Added in version 3.7.

> **Zobacz także**
>
> `os.register_at_fork()` allows registering custom Python functions to be called by `PyOS_BeforeFork()`, `PyOS_AfterFork_Parent()` and `PyOS_AfterFork_Child()`.

void **PyOS_AfterFork**()

Część stabilnego ABI *on platforms with fork().* Function to update some internal state after a process fork; this should be called in the new process if the Python interpreter will continue to be used. If a new executable is loaded into the new process, this function does not need to be called.

Niezalecane od wersji 3.7: This function is superseded by `PyOS_AfterFork_Child()`.

int **PyOS_CheckStack**()

Część stabilnego ABI *on platforms with USE_STACKCHECK od wersji 3.7.* Return true when the interpreter runs out of stack space. This is a reliable check, but is only available when `USE_STACKCHECK` is defined (currently on certain versions of Windows using the Microsoft Visual C++ compiler). `USE_STACKCHECK` will be defined automatically; you should never change the definition in your own code.

typedef void (*__PyOS_sighandler_t__)(int)

Część stabilnego ABI.

PyOS_sighandler_t **PyOS_getsig**(int i)

Część stabilnego ABI. Return the current signal handler for signal *i*. This is a thin wrapper around either `sigaction()` or `signal()`. Do not call those functions directly!

PyOS_sighandler_t **PyOS_setsig**(int i, *PyOS_sighandler_t* h)

Część stabilnego ABI. Set the signal handler for signal *i* to be *h*; return the old signal handler. This is a thin wrapper around either `sigaction()` or `signal()`. Do not call those functions directly!

wchar_t *`Py_DecodeLocale`(const char *arg, size_t *size)

Część stabilnego ABI *od wersji 3.7.*

> **Ostrzeżenie**
>
> This function should not be called directly: use the `PyConfig` API with the `PyConfig_SetBytesString()` function which ensures that *Python is preinitialized*.
>
> This function must not be called before *Python is preinitialized* and so that the LC_CTYPE locale is properly configured: see the `Py_PreInitialize()` function.

Decode a byte string from the *filesystem encoding and error handler*. If the error handler is surrogateescape error handler, undecodable bytes are decoded as characters in range U+DC80..U+DCFF; and if a byte sequence can be decoded as a surrogate character, the bytes are escaped using the surrogateescape error handler instead of decoding them.

Return a pointer to a newly allocated wide character string, use `PyMem_RawFree()` to free the memory. If size is not `NULL`, write the number of wide characters excluding the null character into `*size`

Return `NULL` on decoding error or memory allocation error. If *size* is not `NULL`, `*size` is set to `(size_t)-1` on memory error or set to `(size_t)-2` on decoding error.

The *filesystem encoding and error handler* are selected by `PyConfig_Read()`: see `filesystem_encoding` and `filesystem_errors` members of `PyConfig`.

Decoding errors should never happen, unless there is a bug in the C library.

Use the `Py_EncodeLocale()` function to encode the character string back to a byte string.

> **Zobacz także**
>
> The `PyUnicode_DecodeFSDefaultAndSize()` and `PyUnicode_DecodeLocaleAndSize()` functions.

Added in version 3.5.

Zmienione w wersji 3.7: The function now uses the UTF-8 encoding in the Python UTF-8 Mode.

Zmienione w wersji 3.8: The function now uses the UTF-8 encoding on Windows if `PyPreConfig.legacy_windows_fs_encoding` is zero;

char *`Py_EncodeLocale`(const wchar_t *text, size_t *error_pos)

Część stabilnego ABI *od wersji 3.7.* Encode a wide character string to the *filesystem encoding and error handler*. If the error handler is surrogateescape error handler, surrogate characters in the range U+DC80..U+DCFF are converted to bytes 0x80..0xFF.

Return a pointer to a newly allocated byte string, use `PyMem_Free()` to free the memory. Return `NULL` on encoding error or memory allocation error.

If error_pos is not `NULL`, `*error_pos` is set to `(size_t)-1` on success, or set to the index of the invalid character on encoding error.

The *filesystem encoding and error handler* are selected by `PyConfig_Read()`: see `filesystem_encoding` and `filesystem_errors` members of `PyConfig`.

Use the `Py_DecodeLocale()` function to decode the bytes string back to a wide character string.

> **Ostrzeżenie**
>
> This function must not be called before *Python is preinitialized* and so that the LC_CTYPE locale is properly configured: see the `Py_PreInitialize()` function.

Zobacz także

The `PyUnicode_EncodeFSDefault()` and `PyUnicode_EncodeLocale()` functions.

Added in version 3.5.

Zmienione w wersji 3.7: The function now uses the UTF-8 encoding in the Python UTF-8 Mode.

Zmienione w wersji 3.8: The function now uses the UTF-8 encoding on Windows if `PyPreConfig.legacy_windows_fs_encoding` is zero.

6.2 System Functions

These are utility functions that make functionality from the `sys` module accessible to C code. They all work with the current interpreter thread's `sys` module's dict, which is contained in the internal thread state structure.

PyObject ***PySys_GetObject**(const char *name)

Wartość zwracana: pożyczona referencja. Część stabilnego ABI. Return the object *name* from the `sys` module or `NULL` if it does not exist, without setting an exception.

int **PySys_SetObject**(const char *name, *PyObject* *v)

Część stabilnego ABI. Set *name* in the `sys` module to *v* unless *v* is `NULL`, in which case *name* is deleted from the sys module. Returns `0` on success, `-1` on error.

void **PySys_ResetWarnOptions**()

Część stabilnego ABI. Reset `sys.warnoptions` to an empty list. This function may be called prior to `Py_Initialize()`.

Deprecated since version 3.13, will be removed in version 3.15: Clear `sys.warnoptions` and `warnings.filters` instead.

void **PySys_WriteStdout**(const char *format, ...)

Część stabilnego ABI. Write the output string described by *format* to `sys.stdout`. No exceptions are raised, even if truncation occurs (see below).

format should limit the total size of the formatted output string to 1000 bytes or less – after 1000 bytes, the output string is truncated. In particular, this means that no unrestricted „%s" formats should occur; these should be limited using „%.<N>s" where <N> is a decimal number calculated so that <N> plus the maximum size of other formatted text does not exceed 1000 bytes. Also watch out for „%f", which can print hundreds of digits for very large numbers.

If a problem occurs, or `sys.stdout` is unset, the formatted message is written to the real (C level) *stdout*.

void **PySys_WriteStderr**(const char *format, ...)

Część stabilnego ABI. As `PySys_WriteStdout()`, but write to `sys.stderr` or *stderr* instead.

void **PySys_FormatStdout**(const char *format, ...)

Część stabilnego ABI. Function similar to PySys_WriteStdout() but format the message using `PyUnicode_FromFormatV()` and don't truncate the message to an arbitrary length.

Added in version 3.2.

void **PySys_FormatStderr**(const char *format, ...)

Część stabilnego ABI. As `PySys_FormatStdout()`, but write to `sys.stderr` or *stderr* instead.

Added in version 3.2.

PyObject ***PySys_GetXOptions**()

Wartość zwracana: pożyczona referencja. Część stabilnego ABI *od wersji 3.7.* Return the current dictionary of `-X` options, similarly to `sys._xoptions`. On error, `NULL` is returned and an exception is set.

Added in version 3.2.

int **PySys_Audit**(const char *event, const char *format, ...)

Część stabilnego ABI *od wersji 3.13.* Raise an auditing event with any active hooks. Return zero for success and non-zero with an exception set on failure.

The *event* string argument must not be *NULL*.

If any hooks have been added, *format* and other arguments will be used to construct a tuple to pass. Apart from `N`, the same format characters as used in `Py_BuildValue()` are available. If the built value is not a tuple, it will be added into a single-element tuple.

The `N` format option must not be used. It consumes a reference, but since there is no way to know whether arguments to this function will be consumed, using it may cause reference leaks.

Note that `#` format characters should always be treated as `Py_ssize_t`, regardless of whether `PY_SSIZE_T_CLEAN` was defined.

`sys.audit()` performs the same function from Python code.

See also `PySys_AuditTuple()`.

Added in version 3.8.

Zmienione w wersji 3.8.2: Require `Py_ssize_t` for `#` format characters. Previously, an unavoidable deprecation warning was raised.

int **PySys_AuditTuple**(const char *event, *PyObject* *args)

Część stabilnego ABI *od wersji 3.13.* Similar to `PySys_Audit()`, but pass arguments as a Python object. *args* must be a `tuple`. To pass no arguments, *args* can be *NULL*.

Added in version 3.13.

int **PySys_AddAuditHook**(*Py_AuditHookFunction* hook, void *userData)

Append the callable *hook* to the list of active auditing hooks. Return zero on success and non-zero on failure. If the runtime has been initialized, also set an error on failure. Hooks added through this API are called for all interpreters created by the runtime.

The *userData* pointer is passed into the hook function. Since hook functions may be called from different runtimes, this pointer should not refer directly to Python state.

This function is safe to call before `Py_Initialize()`. When called after runtime initialization, existing audit hooks are notified and may silently abort the operation by raising an error subclassed from `Exception` (other errors will not be silenced).

The hook function is always called with the GIL held by the Python interpreter that raised the event.

See **PEP 578** for a detailed description of auditing. Functions in the runtime and standard library that raise events are listed in the audit events table. Details are in each function's documentation.

If the interpreter is initialized, this function raises an auditing event `sys.addaudithook` with no arguments. If any existing hooks raise an exception derived from `Exception`, the new hook will not be added and the exception is cleared. As a result, callers cannot assume that their hook has been added unless they control all existing hooks.

typedef int (***Py_AuditHookFunction**)(const char *event, *PyObject* *args, void *userData)

The type of the hook function. *event* is the C string event argument passed to `PySys_Audit()` or `PySys_AuditTuple()`. *args* is guaranteed to be a `PyTupleObject`. *userData* is the argument passed to PySys_AddAuditHook().

Added in version 3.8.

6.3 Process Control

void **Py_FatalError**(const char *message)

Część stabilnego ABI. Print a fatal error message and kill the process. No cleanup is performed. This function should only be invoked when a condition is detected that would make it dangerous to continue using the Python interpreter; e.g., when the object administration appears to be corrupted. On Unix, the standard C library function `abort()` is called which will attempt to produce a `core` file.

The `Py_FatalError()` function is replaced with a macro which logs automatically the name of the current function, unless the `Py_LIMITED_API` macro is defined.

Zmienione w wersji 3.9: Log the function name automatically.

void **Py_Exit**(int status)

Część stabilnego ABI. Exit the current process. This calls `Py_FinalizeEx()` and then calls the standard C library function `exit(status)`. If `Py_FinalizeEx()` indicates an error, the exit status is set to 120.

Zmienione w wersji 3.6: Errors from finalization no longer ignored.

int **Py_AtExit**(void (*func)())

Część stabilnego ABI. Register a cleanup function to be called by `Py_FinalizeEx()`. The cleanup function will be called with no arguments and should return no value. At most 32 cleanup functions can be registered. When the registration is successful, `Py_AtExit()` returns `0`; on failure, it returns `-1`. The cleanup function registered last is called first. Each cleanup function will be called at most once. Since Python's internal finalization will have completed before the cleanup function, no Python APIs should be called by *func*.

6.4 Importing Modules

PyObject ***PyImport_ImportModule**(const char *name)

Wartość zwracana: nowa referencja. Część stabilnego ABI. This is a wrapper around `PyImport_Import()` which takes a `const char*` as an argument instead of a `PyObject*`.

PyObject ***PyImport_ImportModuleNoBlock**(const char *name)

Wartość zwracana: nowa referencja. Część stabilnego ABI. This function is a deprecated alias of `PyImport_ImportModule()`.

Zmienione w wersji 3.3: This function used to fail immediately when the import lock was held by another thread. In Python 3.3 though, the locking scheme switched to per-module locks for most purposes, so this function's special behaviour isn't needed anymore.

Deprecated since version 3.13, will be removed in version 3.15: Use `PyImport_ImportModule()` instead.

PyObject ***PyImport_ImportModuleEx**(const char *name, *PyObject* *globals, *PyObject* *locals, *PyObject* *fromlist)

Wartość zwracana: nowa referencja. Import a module. This is best described by referring to the built-in Python function `__import__()`.

The return value is a new reference to the imported module or top-level package, or `NULL` with an exception set on failure. Like for `__import__()`, the return value when a submodule of a package was requested is normally the top-level package, unless a non-empty *fromlist* was given.

Failing imports remove incomplete module objects, like with `PyImport_ImportModule()`.

PyObject ***PyImport_ImportModuleLevelObject**(*PyObject* *name, *PyObject* *globals, *PyObject* *locals, *PyObject* *fromlist, int level)

Wartość zwracana: nowa referencja. Część stabilnego ABI *od wersji 3.7.* Import a module. This is best described by referring to the built-in Python function `__import__()`, as the standard `__import__()` function calls this function directly.

The return value is a new reference to the imported module or top-level package, or `NULL` with an exception set on failure. Like for `__import__()`, the return value when a submodule of a package was requested is normally the top-level package, unless a non-empty *fromlist* was given.

Added in version 3.3.

PyObject ***PyImport_ImportModuleLevel**(const char *name, *PyObject* *globals, *PyObject* *locals, *PyObject* *fromlist, int level)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Similar to `PyImport_ImportModuleLevelObject()`, but the name is a UTF-8 encoded string instead of a Unicode object.

Zmienione w wersji 3.3: Negative values for *level* are no longer accepted.

PyObject ***PyImport_Import**(*PyObject* *name)

Wartość zwracana: nowa referencja. Część stabilnego ABI. This is a higher-level interface that calls the current „import hook function" (with an explicit *level* of 0, meaning absolute import). It invokes the `__import__()` function from the `__builtins__` of the current globals. This means that the import is done using whatever import hooks are installed in the current environment.

This function always uses absolute imports.

PyObject ***PyImport_ReloadModule**(*PyObject* *m)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Reload a module. Return a new reference to the reloaded module, or `NULL` with an exception set on failure (the module still exists in this case).

PyObject ***PyImport_AddModuleRef**(const char *name)

Wartość zwracana: nowa referencja. Część stabilnego ABI *od wersji 3.13.* Return the module object corresponding to a module name.

The *name* argument may be of the form `package.module`. First check the modules dictionary if there's one there, and if not, create a new one and insert it in the modules dictionary.

Return a *strong reference* to the module on success. Return `NULL` with an exception set on failure.

The module name *name* is decoded from UTF-8.

This function does not load or import the module; if the module wasn't already loaded, you will get an empty module object. Use `PyImport_ImportModule()` or one of its variants to import a module. Package structures implied by a dotted name for *name* are not created if not already present.

Added in version 3.13.

PyObject ***PyImport_AddModuleObject**(*PyObject* *name)

Wartość zwracana: pożyczona referencja. Część stabilnego ABI *od wersji 3.7.* Similar to `PyImport_AddModuleRef()`, but return a *borrowed reference* and *name* is a Python `str` object.

Added in version 3.3.

PyObject ***PyImport_AddModule**(const char *name)

Wartość zwracana: pożyczona referencja. Część stabilnego ABI. Similar to `PyImport_AddModuleRef()`, but return a *borrowed reference*.

PyObject ***PyImport_ExecCodeModule**(const char *name, *PyObject* *co)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Given a module name (possibly of the form `package.module`) and a code object read from a Python bytecode file or obtained from the built-in function `compile()`, load the module. Return a new reference to the module object, or `NULL` with an exception set if an error occurred. *name* is removed from `sys.modules` in error cases, even if *name* was already in `sys.modules` on entry to `PyImport_ExecCodeModule()`. Leaving incompletely initialized modules in `sys.modules` is dangerous, as imports of such modules have no way to know that the module object is an unknown (and probably damaged with respect to the module author's intents) state.

The module's `__spec__` and `__loader__` will be set, if not set already, with the appropriate values. The spec's loader will be set to the module's `__loader__` (if set) and to an instance of `SourceFileLoader` otherwise.

The module's `__file__` attribute will be set to the code object's `co_filename`. If applicable, `__cached__` will also be set.

This function will reload the module if it was already imported. See `PyImport_ReloadModule()` for the intended way to reload a module.

If *name* points to a dotted name of the form `package.module`, any package structures not already created will still not be created.

See also `PyImport_ExecCodeModuleEx()` and `PyImport_ExecCodeModuleWithPathnames()`.

Zmienione w wersji 3.12: The setting of `__cached__` and `__loader__` is deprecated. See `ModuleSpec` for alternatives.

PyObject ***PyImport_ExecCodeModuleEx**(const char *name, *PyObject* *co, const char *pathname)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Like `PyImport_ExecCodeModule()`, but the `__file__` attribute of the module object is set to *pathname* if it is non-`NULL`.

See also `PyImport_ExecCodeModuleWithPathnames()`.

PyObject ***PyImport_ExecCodeModuleObject**(*PyObject* *name, *PyObject* *co, *PyObject* *pathname, *PyObject* *cpathname)

Wartość zwracana: nowa referencja. Część stabilnego ABI *od wersji 3.7.* Like `PyImport_ExecCodeModuleEx()`, but the `__cached__` attribute of the module object is set to *cpathname* if it is non-`NULL`. Of the three functions, this is the preferred one to use.

Added in version 3.3.

Zmienione w wersji 3.12: Setting `__cached__` is deprecated. See `ModuleSpec` for alternatives.

PyObject ***PyImport_ExecCodeModuleWithPathnames**(const char *name, *PyObject* *co, const char *pathname, const char *cpathname)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Like `PyImport_ExecCodeModuleObject()`, but *name*, *pathname* and *cpathname* are UTF-8 encoded strings. Attempts are also made to figure out what the value for *pathname* should be from *cpathname* if the former is set to `NULL`.

Added in version 3.2.

Zmienione w wersji 3.3: Uses `imp.source_from_cache()` in calculating the source path if only the bytecode path is provided.

Zmienione w wersji 3.12: No longer uses the removed `imp` module.

long **PyImport_GetMagicNumber**()

Część stabilnego ABI. Return the magic number for Python bytecode files (a.k.a. `.pyc` file). The magic number should be present in the first four bytes of the bytecode file, in little-endian byte order. Returns `-1` on error.

Zmienione w wersji 3.3: Return value of `-1` upon failure.

const char ***PyImport_GetMagicTag**()

Część stabilnego ABI. Return the magic tag string for **PEP 3147** format Python bytecode file names. Keep in mind that the value at `sys.implementation.cache_tag` is authoritative and should be used instead of this function.

Added in version 3.2.

PyObject ***PyImport_GetModuleDict**()

Wartość zwracana: pożyczona referencja. Część stabilnego ABI. Return the dictionary used for the module administration (a.k.a. `sys.modules`). Note that this is a per-interpreter variable.

PyObject ***PyImport_GetModule**(*PyObject* *name)

Wartość zwracana: nowa referencja. Część stabilnego ABI *od wersji 3.8.* Return the already imported module with the given name. If the module has not been imported yet then returns `NULL` but does not set an error. Returns `NULL` and sets an error if the lookup failed.

Added in version 3.7.

PyObject ***PyImport_GetImporter*** (*PyObject* *path)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return a finder object for a `sys.path`/`pkg.__path__` item *path*, possibly by fetching it from the `sys.path_importer_cache` dict. If it wasn't yet cached, traverse `sys.path_hooks` until a hook is found that can handle the path item. Return `None` if no hook could; this tells our caller that the *path based finder* could not find a finder for this path item. Cache the result in `sys.path_importer_cache`. Return a new reference to the finder object.

int ***PyImport_ImportFrozenModuleObject*** (*PyObject* *name)

Część stabilnego ABI *od wersji 3.7*. Load a frozen module named *name*. Return `1` for success, `0` if the module is not found, and `-1` with an exception set if the initialization failed. To access the imported module on a successful load, use `PyImport_ImportModule()`. (Note the misnomer — this function would reload the module if it was already imported.)

Added in version 3.3.

Zmienione w wersji 3.4: The `__file__` attribute is no longer set on the module.

int ***PyImport_ImportFrozenModule*** (const char *name)

Część stabilnego ABI. Similar to `PyImport_ImportFrozenModuleObject()`, but the name is a UTF-8 encoded string instead of a Unicode object.

struct ***_frozen***

This is the structure type definition for frozen module descriptors, as generated by the ***freeze*** utility (see `Tools/freeze/` in the Python source distribution). Its definition, found in `Include/import.h`, is:

```
struct _frozen {
    const char *name;
    const unsigned char *code;
    int size;
    bool is_package;
};
```

Zmienione w wersji 3.11: The new `is_package` field indicates whether the module is a package or not. This replaces setting the `size` field to a negative value.

const struct *_frozen* ***PyImport_FrozenModules***

This pointer is initialized to point to an array of `_frozen` records, terminated by one whose members are all `NULL` or zero. When a frozen module is imported, it is searched in this table. Third-party code could play tricks with this to provide a dynamically created collection of frozen modules.

int ***PyImport_AppendInittab*** (const char *name, *PyObject* *(*initfunc)(void))

Część stabilnego ABI. Add a single module to the existing table of built-in modules. This is a convenience wrapper around `PyImport_ExtendInittab()`, returning `-1` if the table could not be extended. The new module can be imported by the name *name*, and uses the function *initfunc* as the initialization function called on the first attempted import. This should be called before `Py_Initialize()`.

struct ***_inittab***

Structure describing a single entry in the list of built-in modules. Programs which embed Python may use an array of these structures in conjunction with `PyImport_ExtendInittab()` to provide additional built-in modules. The structure consists of two members:

const char ***name***

The module name, as an ASCII encoded string.

PyObject *(***initfunc***)(void)

Initialization function for a module built into the interpreter.

int ***PyImport_ExtendInittab*** (struct *_inittab* *newtab)

Add a collection of modules to the table of built-in modules. The *newtab* array must end with a sentinel entry which contains `NULL` for the `name` field; failure to provide the sentinel value can result in a memory fault. Returns `0` on success or `-1` if insufficient memory could be allocated to extend the internal table. In the event of failure, no modules are added to the internal table. This must be called before `Py_Initialize()`.

If Python is initialized multiple times, `PyImport_AppendInittab()` or `PyImport_ExtendInittab()` must be called before each Python initialization.

6.5 Data marshalling support

These routines allow C code to work with serialized objects using the same data format as the `marshal` module. There are functions to write data into the serialization format, and additional functions that can be used to read the data back. Files used to store marshalled data must be opened in binary mode.

Numeric values are stored with the least significant byte first.

The module supports two versions of the data format: version 0 is the historical version, version 1 shares interned strings in the file, and upon unmarshalling. Version 2 uses a binary format for floating-point numbers. `Py_MARSHAL_VERSION` indicates the current file format (currently 2).

void **PyMarshal_WriteLongToFile**(long value, FILE *file, int version)

Marshal a `long` integer, *value*, to *file*. This will only write the least-significant 32 bits of *value*; regardless of the size of the native `long` type. *version* indicates the file format.

This function can fail, in which case it sets the error indicator. Use `PyErr_Occurred()` to check for that.

void **PyMarshal_WriteObjectToFile**(*PyObject* *value, FILE *file, int version)

Marshal a Python object, *value*, to *file*. *version* indicates the file format.

This function can fail, in which case it sets the error indicator. Use `PyErr_Occurred()` to check for that.

PyObject ***PyMarshal_WriteObjectToString**(*PyObject* *value, int version)

Wartość zwracana: nowa referencja. Return a bytes object containing the marshalled representation of *value*. *version* indicates the file format.

The following functions allow marshalled values to be read back in.

long **PyMarshal_ReadLongFromFile**(FILE *file)

Return a C `long` from the data stream in a `FILE*` opened for reading. Only a 32-bit value can be read in using this function, regardless of the native size of `long`.

On error, sets the appropriate exception (`EOFError`) and returns `-1`.

int **PyMarshal_ReadShortFromFile**(FILE *file)

Return a C `short` from the data stream in a `FILE*` opened for reading. Only a 16-bit value can be read in using this function, regardless of the native size of `short`.

On error, sets the appropriate exception (`EOFError`) and returns `-1`.

PyObject ***PyMarshal_ReadObjectFromFile**(FILE *file)

Wartość zwracana: nowa referencja. Return a Python object from the data stream in a `FILE*` opened for reading.

On error, sets the appropriate exception (`EOFError`, `ValueError` or `TypeError`) and returns `NULL`.

PyObject ***PyMarshal_ReadLastObjectFromFile**(FILE *file)

Wartość zwracana: nowa referencja. Return a Python object from the data stream in a `FILE*` opened for reading. Unlike `PyMarshal_ReadObjectFromFile()`, this function assumes that no further objects will be read from the file, allowing it to aggressively load file data into memory so that the de-serialization can operate from data in memory rather than reading a byte at a time from the file. Only use these variant if you are certain that you won't be reading anything else from the file.

On error, sets the appropriate exception (`EOFError`, `ValueError` or `TypeError`) and returns `NULL`.

PyObject ***PyMarshal_ReadObjectFromString**(const char *data, *Py_ssize_t* len)

Wartość zwracana: nowa referencja. Return a Python object from the data stream in a byte buffer containing *len* bytes pointed to by *data*.

On error, sets the appropriate exception (`EOFError`, `ValueError` or `TypeError`) and returns `NULL`.

6.6 Pobieranie kolejnych rzeczy podanych na wejściu i konstruowanie wartości.

Te działania są użyteczne przy tworzeniu swoich własnych zadań rozszerzających i rozszerzających sposobów działania. Dodatkowe informacje i przykłady dostępne są w extending-index.

The first three of these functions described, `PyArg_ParseTuple()`, `PyArg_ParseTupleAndKeywords()`, and `PyArg_Parse()`, all use *format strings* which are used to tell the function about the expected arguments. The format strings use the same syntax for each of these functions.

6.6.1 Parsowanie argumentów

napis kształtujący może być pusty lub składać się z „jednostek kształtujących". Jednostka ukształtowania opisuje jeden pytonowski przedmiot; jest to zazwyczaj jedna litera, lub kolejką jednostek ukształtowania w ujętych w nawias. Z kilkoma wyjątkami, jednostka ukształtowania, która nie jest kolejką jednostek ujętą w nawias zwykle odpowiada pojedynczo umiejscowionej rzeczy przekazywanej dla tych zadań. W poniższych zapisach cytat jest jednostką kształtującą; polecenie ujęte w nawias okrągły ( ) oznacza typ przedmiotu w języku pytonowskim, który odpowiada jednostce kształtującej, a zapis ujęty w nawiasie kwadratowym [ ] określa typ przedmiotu / przedmiotów w języku C, którego miejsce powinno zostać wskazane i przekazane dla zadania.

Napisy i skrzynki wymiany

> **Informacja**
>
> On Python 3.12 and older, the macro `PY_SSIZE_T_CLEAN` must be defined before including `Python.h` to use all `#` variants of formats (`s#`, `y#`, etc.) explained below. This is not necessary on Python 3.13 and later.

Formaty te umożliwiają dostęp do obiektu jako ciągłego fragmentu pamięci. Nie musisz zapewniać nieprzetworzonej pamięci dla zwróconego obszaru unicode lub bajtów.

Bufory nie są zakończone znakiem NULL, chyba że zaznaczono inaczej.

Istnieją trzy sposoby konwersji łańcuchów i buforów na język C:

- Formats such as `y*` and `s*` fill a `Py_buffer` structure. This locks the underlying buffer so that the caller can subsequently use the buffer even inside a `Py_BEGIN_ALLOW_THREADS` block without the risk of mutable data being resized or destroyed. As a result, **you have to call** `PyBuffer_Release()` after you have finished processing the data (or in any early abort case).
- The `es`, `es#`, `et` and `et#` formats allocate the result buffer. **You have to call** `PyMem_Free()` after you have finished processing the data (or in any early abort case).
- Other formats take a `str` or a read-only *bytes-like object*, such as `bytes`, and provide a `const char *` pointer to its buffer. In this case the buffer is „borrowed": it is managed by the corresponding Python object, and shares the lifetime of this object. You won't have to release any memory yourself.

  To ensure that the underlying buffer may be safely borrowed, the object's `PyBufferProcs.bf_releasebuffer` field must be `NULL`. This disallows common mutable objects such as `bytearray`, but also some read-only objects such as `memoryview` of `bytes`.

  Besides this `bf_releasebuffer` requirement, there is no check to verify whether the input object is immutable (e.g. whether it would honor a request for a writable buffer, or whether another thread can mutate the data).

`s` (`str`) [const char *]
Convert a Unicode object to a C pointer to a character string. A pointer to an existing string is stored in the character pointer variable whose address you pass. The C string is NUL-terminated. The Python string must not contain embedded null code points; if it does, a `ValueError` exception is raised. Unicode objects are converted to C strings using `'utf-8'` encoding. If this conversion fails, a `UnicodeError` is raised.

> **Informacja**
>
> This format does not accept *bytes-like objects*. If you want to accept filesystem paths and convert them to C character strings, it is preferable to use the `O&` format with `PyUnicode_FSConverter()` as *converter*.

Zmienione w wersji 3.5: Previously, `TypeError` was raised when embedded null code points were encountered in the Python string.

`s*` (`str` or *bytes-like object*) [Py_buffer]
This format accepts Unicode objects as well as bytes-like objects. It fills a `Py_buffer` structure provided by the caller. In this case the resulting C string may contain embedded NUL bytes. Unicode objects are converted to C strings using `'utf-8'` encoding.

`s#` (`str`, read-only *bytes-like object*) [const char *, `Py_ssize_t`]
Like `s*`, except that it provides a *borrowed buffer*. The result is stored into two C variables, the first one a pointer to a C string, the second one its length. The string may contain embedded null bytes. Unicode objects are converted to C strings using `'utf-8'` encoding.

`z` (`str` or `None`) [const char *]
Like `s`, but the Python object may also be `None`, in which case the C pointer is set to `NULL`.

`z*` (`str`, *bytes-like object* or `None`) [Py_buffer]
Like `s*`, but the Python object may also be `None`, in which case the `buf` member of the `Py_buffer` structure is set to `NULL`.

`z#` (`str`, read-only *bytes-like object* or `None`) [const char *, `Py_ssize_t`]
Like `s#`, but the Python object may also be `None`, in which case the C pointer is set to `NULL`.

`y` (read-only *bytes-like object*) [const char *]
This format converts a bytes-like object to a C pointer to a *borrowed* character string; it does not accept Unicode objects. The bytes buffer must not contain embedded null bytes; if it does, a `ValueError` exception is raised.

Zmienione w wersji 3.5: Previously, `TypeError` was raised when embedded null bytes were encountered in the bytes buffer.

`y*` (*bytes-like object*) [Py_buffer]
This variant on `s*` doesn't accept Unicode objects, only bytes-like objects. **This is the recommended way to accept binary data.**

`y#` (read-only *bytes-like object*) [const char *, `Py_ssize_t`]
Ten wariant `s#` nie akceptuje obiektów Unicode, a jedynie bajto-podobne obiekty.

`S` (`bytes`) [PyBytesObject *]
Requires that the Python object is a `bytes` object, without attempting any conversion. Raises `TypeError` if the object is not a bytes object. The C variable may also be declared as `PyObject*`.

`Y` (`bytearray`) [PyByteArrayObject *]
Requires that the Python object is a `bytearray` object, without attempting any conversion. Raises `TypeError` if the object is not a `bytearray` object. The C variable may also be declared as `PyObject*`.

`U` (`str`) [PyObject *]
Requires that the Python object is a Unicode object, without attempting any conversion. Raises `TypeError` if the object is not a Unicode object. The C variable may also be declared as `PyObject*`.

`w*` (read-write *bytes-like object*) [Py_buffer]
This format accepts any object which implements the read-write buffer interface. It fills a `Py_buffer` structure provided by the caller. The buffer may contain embedded null bytes. The caller have to call `PyBuffer_Release()` when it is done with the buffer.

`es` (`str`) [const char *encoding, char **buffer]
Ten wariant `s` jest używany do zakodowania Unicode w buforze znaków. To działa tylko dla zakodowanych danych bez osadzonych znaków NUL.

This format requires two arguments. The first is only used as input, and must be a `const char*` which points to the name of an encoding as a NUL-terminated string, or `NULL`, in which case `'utf-8'` encoding is used.

An exception is raised if the named encoding is not known to Python. The second argument must be a `char**`; the value of the pointer it references will be set to a buffer with the contents of the argument text. The text will be encoded in the encoding specified by the first argument.

`PyArg_ParseTuple()` will allocate a buffer of the needed size, copy the encoded data into this buffer and adjust **buffer* to reference the newly allocated storage. The caller is responsible for calling `PyMem_Free()` to free the allocated buffer after use.

et (str, bytes or bytearray) [const char *encoding, char **buffer]
Podobnie jak `es` z wyjątkiem tego, że obiekty ciągów znaków są przekazywane dalej bez ich zapisywania. Zamiast tego implementacja zakłada, że obiekt łańcucha znaków wykorzystuje kodowanie przekazane jako parametr.

es# (str) [const char *encoding, char **buffer, Py_ssize_t *buffer_length]
Ten wariant `s#` używany jest do kodowania Unicode w buforze znaków. W przeciwieństwie do formatu `es`, ten wariant pozwala wprowadzać dane zawierające znaki NUL.

It requires three arguments. The first is only used as input, and must be a `const char*` which points to the name of an encoding as a NUL-terminated string, or `NULL`, in which case `'utf-8'` encoding is used. An exception is raised if the named encoding is not known to Python. The second argument must be a `char**`; the value of the pointer it references will be set to a buffer with the contents of the argument text. The text will be encoded in the encoding specified by the first argument. The third argument must be a pointer to an integer; the referenced integer will be set to the number of bytes in the output buffer.

Istnieją dwa tryby pracy:

If **buffer* points a `NULL` pointer, the function will allocate a buffer of the needed size, copy the encoded data into this buffer and set **buffer* to reference the newly allocated storage. The caller is responsible for calling `PyMem_Free()` to free the allocated buffer after usage.

If **buffer* points to a non-`NULL` pointer (an already allocated buffer), `PyArg_ParseTuple()` will use this location as the buffer and interpret the initial value of **buffer_length* as the buffer size. It will then copy the encoded data into the buffer and NUL-terminate it. If the buffer is not large enough, a `ValueError` will be set.

W obu przypadkach, **buffer_length* jest ustawiany na długość zakodowanych danych z pominięciem zakańczającego znaku NUL.

et# (str, bytes or bytearray) [const char *encoding, char **buffer, Py_ssize_t *buffer_length]
Tak samo, jak w `es#` oprócz tego, że obiekty ciągu bajtów są przekazywane do funkcji bez ich zapisywania. Zamiast tego, implementacja zakłada że obiekt ciągu bajtów używa kodowania przekazywanego w parametrze.

Zmienione w wersji 3.12: `u`, `u#`, `Z`, and `Z#` are removed because they used a legacy `Py_UNICODE*` representation.

Liczby

b (int) [unsigned char]
Convert a nonnegative Python integer to an unsigned tiny int, stored in a C `unsigned char`.

B (int) [unsigned char]
Convert a Python integer to a tiny int without overflow checking, stored in a C `unsigned char`.

h (int) [short int]
Convert a Python integer to a C `short int`.

H (int) [unsigned short int]
Convert a Python integer to a C `unsigned short int`, without overflow checking.

i (int) [int]
Convert a Python integer to a plain C `int`.

I (int) [unsigned int]
Convert a Python integer to a C `unsigned int`, without overflow checking.

l (int) [long int]
Convert a Python integer to a C `long int`.

k (int) [unsigned long]
Convert a Python integer to a C `unsigned long` without overflow checking.

L (int) [long long]
Convert a Python integer to a C `long long`.

K (int) [unsigned long long]
Convert a Python integer to a C `unsigned long long` without overflow checking.

n (int) [*Py_ssize_t*]
Convert a Python integer to a C *Py_ssize_t*.

c (bytes or bytearray of length 1) [char]
Convert a Python byte, represented as a `bytes` or `bytearray` object of length 1, to a C `char`.

Zmienione w wersji 3.3: Allow `bytearray` objects.

C (str of length 1) [int]
Convert a Python character, represented as a `str` object of length 1, to a C `int`.

f (float) [float]
Convert a Python floating-point number to a C `float`.

d (float) [double]
Convert a Python floating-point number to a C `double`.

D (complex) [Py_complex]
Convert a Python complex number to a C *Py_complex* structure.

Inne obiekty

O (object) [PyObject *]
Store a Python object (without any conversion) in a C object pointer. The C program thus receives the actual object that was passed. A new *strong reference* to the object is not created (i.e. its reference count is not increased). The pointer stored is not `NULL`.

O! (object) [*typeobject*, PyObject *]
Store a Python object in a C object pointer. This is similar to `O`, but takes two C arguments: the first is the address of a Python type object, the second is the address of the C variable (of type *PyObject**) into which the object pointer is stored. If the Python object does not have the required type, `TypeError` is raised.

O& (object) [*converter*, *anything*]
Convert a Python object to a C variable through a *converter* function. This takes two arguments: the first is a function, the second is the address of a C variable (of arbitrary type), converted to `void*`. The *converter* function in turn is called as follows:

```
status = converter(object, address);
```

where *object* is the Python object to be converted and *address* is the `void*` argument that was passed to the `PyArg_Parse*` function. The returned *status* should be `1` for a successful conversion and `0` if the conversion has failed. When the conversion fails, the *converter* function should raise an exception and leave the content of *address* unmodified.

If the *converter* returns `Py_CLEANUP_SUPPORTED`, it may get called a second time if the argument parsing eventually fails, giving the converter a chance to release any memory that it had already allocated. In this second call, the *object* parameter will be `NULL`; *address* will have the same value as in the original call.

Zmienione w wersji 3.1: `Py_CLEANUP_SUPPORTED` został dodany.

p (bool) [int]
Tests the value passed in for truth (a boolean **p**redicate) and converts the result to its equivalent C true/false integer value. Sets the int to `1` if the expression was true and `0` if it was false. This accepts any valid Python value. See truth for more information about how Python tests values for truth.

Added in version 3.3.

(items) (tuple) [*matching-items*]
Obiekt musi być sekwencją języka Python, której długość jest liczbą elementów formatu mierzoną w *elementach*. Argumenty języka C muszą odpowiadać poszczególnym jednostkom formatu w *elementach*. Jednostki formatu dla sekwencji mogą być zagnieżdżane.

It is possible to pass „long" integers (integers whose value exceeds the platform's `LONG_MAX`) however no proper range checking is done — the most significant bits are silently truncated when the receiving field is too small to receive the value (actually, the semantics are inherited from downcasts in C — your mileage may vary).

Kilka innych znaków ma jeszcze znaczenie w ciągu formatu. Nie mogą one wystąpić wewnątrz zagnieżdżonych nawiasach okrągłych. Są to:

|
Indicates that the remaining arguments in the Python argument list are optional. The C variables corresponding to optional arguments should be initialized to their default value — when an optional argument is not specified, `PyArg_ParseTuple()` does not touch the contents of the corresponding C variable(s).

$
`PyArg_ParseTupleAndKeywords()` only: Indicates that the remaining arguments in the Python argument list are keyword-only. Currently, all keyword-only arguments must also be optional arguments, so `|` must always be specified before `$` in the format string.

Added in version 3.3.

:
The list of format units ends here; the string after the colon is used as the function name in error messages (the „associated value" of the exception that `PyArg_ParseTuple()` raises).

;
Na tym kończy się lista jednostek formatu; ciąg po średniku jest traktowany jako informacja o błędzie do użycia *zamiast* automatycznej wiadomości o błędzie. Znaki dwukropka `:` i średnika `;` wzajemnie się wykluczają.

Note that any Python object references which are provided to the caller are *borrowed* references; do not release them (i.e. do not decrement their reference count)!

Dodatkowe parametry przekazywane do tych funkcji muszą być adresami zmiennych których typ jest określany przez ciąg formatu; są one używane do przechowywania wartości z krotki wejściowej. Jest parę przypadków, jak opisuje to lista jednostek formatu powyżej, gdzie te parametry są używane jako wprowadzane wartości; w takich przypadku powinny one odpowiadać temu, co jest określone we właściwych im jednostach formatu.

For the conversion to succeed, the *arg* object must match the format and the format must be exhausted. On success, the `PyArg_Parse*` functions return true, otherwise they return false and raise an appropriate exception. When the `PyArg_Parse*` functions fail due to conversion failure in one of the format units, the variables at the addresses corresponding to that and the following format units are left untouched.

Funkcje interfejsu programowania aplikacji

int **PyArg_ParseTuple**(*PyObject* *args, const char *format, ...)
Część stabilnego ABI. Parse the parameters of a function that takes only positional parameters into local variables. Returns true on success; on failure, it returns false and raises the appropriate exception.

int **PyArg_VaParse**(*PyObject* *args, const char *format, va_list vargs)
Część stabilnego ABI. Identical to `PyArg_ParseTuple()`, except that it accepts a va_list rather than a variable number of arguments.

int **PyArg_ParseTupleAndKeywords**(*PyObject* *args, *PyObject* *kw, const char *format, char *const *keywords, ...)
Część stabilnego ABI. Parse the parameters of a function that takes both positional and keyword parameters into local variables. The *keywords* argument is a `NULL`-terminated array of keyword parameter names specified as null-terminated ASCII or UTF-8 encoded C strings. Empty names denote *positional-only parameters*. Returns true on success; on failure, it returns false and raises the appropriate exception.

> **Informacja**
>
> The *keywords* parameter declaration is `char *const*` in C and `const char *const*` in C++. This can be overridden with the `PY_CXX_CONST` macro.

Zmienione w wersji 3.6: Added support for *positional-only parameters*.

Zmienione w wersji 3.13: The *keywords* parameter has now type `char *const*` in C and `const char *const*` in C++, instead of `char**`. Added support for non-ASCII keyword parameter names.

int **PyArg_VaParseTupleAndKeywords**(*PyObject* *args, *PyObject* *kw, const char *format, char *const *keywords, va_list vargs)

Część stabilnego ABI. Identical to `PyArg_ParseTupleAndKeywords()`, except that it accepts a va_list rather than a variable number of arguments.

int **PyArg_ValidateKeywordArguments**(*PyObject**)

Część stabilnego ABI. Ensure that the keys in the keywords argument dictionary are strings. This is only needed if `PyArg_ParseTupleAndKeywords()` is not used, since the latter already does this check.

Added in version 3.2.

int **PyArg_Parse**(*PyObject* *args, const char *format, ...)

Część stabilnego ABI. Parse the parameter of a function that takes a single positional parameter into a local variable. Returns true on success; on failure, it returns false and raises the appropriate exception.

Example:

```
// Function using METH_O calling convention
static PyObject*
my_function(PyObject *module, PyObject *arg)
{
    int value;
    if (!PyArg_Parse(arg, "i:my_function", &value)) {
        return NULL;
    }
    // ... use value ...
}
```

int **PyArg_UnpackTuple**(*PyObject* *args, const char *name, *Py_ssize_t* min, *Py_ssize_t* max, ...)

Część stabilnego ABI. A simpler form of parameter retrieval which does not use a format string to specify the types of the arguments. Functions which use this method to retrieve their parameters should be declared as `METH_VARARGS` in function or method tables. The tuple containing the actual parameters should be passed as *args*; it must actually be a tuple. The length of the tuple must be at least *min* and no more than *max*; *min* and *max* may be equal. Additional arguments must be passed to the function, each of which should be a pointer to a `PyObject*` variable; these will be filled in with the values from *args*; they will contain *borrowed references*. The variables which correspond to optional parameters not given by *args* will not be filled in; these should be initialized by the caller. This function returns true on success and false if *args* is not a tuple or contains the wrong number of elements; an exception will be set if there was a failure.

This is an example of the use of this function, taken from the sources for the `_weakref` helper module for weak references:

```
static PyObject *
weakref_ref(PyObject *self, PyObject *args)
{
    PyObject *object;
    PyObject *callback = NULL;
    PyObject *result = NULL;
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
    if (PyArg_UnpackTuple(args, "ref", 1, 2, &object, &callback)) {
        result = PyWeakref_NewRef(object, callback);
    }
    return result;
}
```

The call to `PyArg_UnpackTuple()` in this example is entirely equivalent to this call to `PyArg_ParseTuple()`:

```
PyArg_ParseTuple(args, "O|O:ref", &object, &callback)
```

PY_CXX_CONST

The value to be inserted, if any, before `char *const*` in the *keywords* parameter declaration of `PyArg_ParseTupleAndKeywords()` and `PyArg_VaParseTupleAndKeywords()`. Default empty for C and `const` for C++ (`const char *const*`). To override, define it to the desired value before including `Python.h`.

Added in version 3.13.

6.6.2 Budowanie wartości

PyObject ***Py_BuildValue**(const char *format, ...)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Create a new value based on a format string similar to those accepted by the `PyArg_Parse*` family of functions and a sequence of values. Returns the value or `NULL` in the case of an error; an exception will be raised if `NULL` is returned.

`Py_BuildValue()` does not always build a tuple. It builds a tuple only if its format string contains two or more format units. If the format string is empty, it returns `None`; if it contains exactly one format unit, it returns whatever object is described by that format unit. To force it to return a tuple of size 0 or one, parenthesize the format string.

When memory buffers are passed as parameters to supply data to build objects, as for the `s` and `s#` formats, the required data is copied. Buffers provided by the caller are never referenced by the objects created by `Py_BuildValue()`. In other words, if your code invokes `malloc()` and passes the allocated memory to `Py_BuildValue()`, your code is responsible for calling `free()` for that memory once `Py_BuildValue()` returns.

In the following description, the quoted form is the format unit; the entry in (round) parentheses is the Python object type that the format unit will return; and the entry in [square] brackets is the type of the C value(s) to be passed.

The characters space, tab, colon and comma are ignored in format strings (but not within format units such as `s#`). This can be used to make long format strings a tad more readable.

s (str or None) [const char *]
Convert a null-terminated C string to a Python `str` object using `'utf-8'` encoding. If the C string pointer is `NULL`, `None` is used.

s# (str or None) [const char *, Py_ssize_t]
Convert a C string and its length to a Python `str` object using `'utf-8'` encoding. If the C string pointer is `NULL`, the length is ignored and `None` is returned.

y (bytes) [const char *]
This converts a C string to a Python `bytes` object. If the C string pointer is `NULL`, `None` is returned.

y# (bytes) [const char *, Py_ssize_t]
This converts a C string and its lengths to a Python object. If the C string pointer is `NULL`, `None` is returned.

z (str or None) [const char *]
Same as `s`.

z# (str or None) [const char *, Py_ssize_t]
Same as s#.

u (str) [const wchar_t *]
Convert a null-terminated wchar_t buffer of Unicode (UTF-16 or UCS-4) data to a Python Unicode object. If the Unicode buffer pointer is NULL, None is returned.

u# (str) [const wchar_t *, Py_ssize_t]
Convert a Unicode (UTF-16 or UCS-4) data buffer and its length to a Python Unicode object. If the Unicode buffer pointer is NULL, the length is ignored and None is returned.

U (str or None) [const char *]
Same as s.

U# (str or None) [const char *, Py_ssize_t]
Same as s#.

i (int) [int]
Convert a plain C int to a Python integer object.

b (int) [char]
Convert a plain C char to a Python integer object.

h (int) [short int]
Convert a plain C short int to a Python integer object.

l (int) [long int]
Convert a C long int to a Python integer object.

B (int) [unsigned char]
Convert a C unsigned char to a Python integer object.

H (int) [unsigned short int]
Convert a C unsigned short int to a Python integer object.

I (int) [unsigned int]
Convert a C unsigned int to a Python integer object.

k (int) [unsigned long]
Convert a C unsigned long to a Python integer object.

L (int) [long long]
Convert a C long long to a Python integer object.

K (int) [unsigned long long]
Convert a C unsigned long long to a Python integer object.

n (int) [Py_ssize_t]
Convert a C Py_ssize_t to a Python integer.

c (bytes of length 1) [char]
Convert a C int representing a byte to a Python bytes object of length 1.

C (str of length 1) [int]
Convert a C int representing a character to Python str object of length 1.

d (float) [double]
Convert a C double to a Python floating-point number.

f (float) [float]
Convert a C float to a Python floating-point number.

D (complex) [Py_complex *]
Convert a C Py_complex structure to a Python complex number.

O (object) [PyObject *]
Pass a Python object untouched but create a new *strong reference* to it (i.e. its reference count is incremented by one). If the object passed in is a NULL pointer, it is assumed that this was caused because the

call producing the argument found an error and set an exception. Therefore, `Py_BuildValue()` will return `NULL` but won't raise an exception. If no exception has been raised yet, `SystemError` is set.

`S` (object) [PyObject *]
Same as `O`.

`N` (object) [PyObject *]
Same as `O`, except it doesn't create a new *strong reference*. Useful when the object is created by a call to an object constructor in the argument list.

`O&` (object) [*converter*, *anything*]
Convert *anything* to a Python object through a *converter* function. The function is called with *anything* (which should be compatible with `void*`) as its argument and should return a „new" Python object, or `NULL` if an error occurred.

`(items)` (tuple) [*matching-items*]
Convert a sequence of C values to a Python tuple with the same number of items.

`[items]` (list) [*matching-items*]
Convert a sequence of C values to a Python list with the same number of items.

`{items}` (dict) [*matching-items*]
Convert a sequence of C values to a Python dictionary. Each pair of consecutive C values adds one item to the dictionary, serving as key and value, respectively.

If there is an error in the format string, the `SystemError` exception is set and `NULL` returned.

PyObject ***Py_VaBuildValue**(const char *format, va_list vargs)
Wartość zwracana: nowa referencja. Część stabilnego ABI. Identical to `Py_BuildValue()`, except that it accepts a va_list rather than a variable number of arguments.

6.7 String conversion and formatting

Functions for number conversion and formatted string output.

int **PyOS_snprintf**(char *str, size_t size, const char *format, ...)
Część stabilnego ABI. Output not more than *size* bytes to *str* according to the format string *format* and the extra arguments. See the Unix man page `snprintf(3)`.

int **PyOS_vsnprintf**(char *str, size_t size, const char *format, va_list va)
Część stabilnego ABI. Output not more than *size* bytes to *str* according to the format string *format* and the variable argument list *va*. Unix man page `vsnprintf(3)`.

`PyOS_snprintf()` and `PyOS_vsnprintf()` wrap the Standard C library functions `snprintf()` and `vsnprintf()`. Their purpose is to guarantee consistent behavior in corner cases, which the Standard C functions do not.

The wrappers ensure that `str[size-1]` is always `'\0'` upon return. They never write more than *size* bytes (including the trailing `'\0'`) into str. Both functions require that `str != NULL`, `size > 0`, `format != NULL` and `size < INT_MAX`. Note that this means there is no equivalent to the C99 `n = snprintf(NULL, 0, ...)` which would determine the necessary buffer size.

The return value (*rv*) for these functions should be interpreted as follows:

- When `0 <= rv < size`, the output conversion was successful and *rv* characters were written to *str* (excluding the trailing `'\0'` byte at `str[rv]`).
- When `rv >= size`, the output conversion was truncated and a buffer with `rv + 1` bytes would have been needed to succeed. `str[size-1]` is `'\0'` in this case.
- When `rv < 0`, „something bad happened." `str[size-1]` is `'\0'` in this case too, but the rest of *str* is undefined. The exact cause of the error depends on the underlying platform.

The following functions provide locale-independent string to number conversions.

unsigned long **PyOS_strtoul**(const char *str, char **ptr, int base)

Część stabilnego ABI. Convert the initial part of the string in `str` to an `unsigned long` value according to the given `base`, which must be between `2` and `36` inclusive, or be the special value `0`.

Leading white space and case of characters are ignored. If `base` is zero it looks for a leading `0b`, `0o` or `0x` to tell which base. If these are absent it defaults to `10`. Base must be 0 or between 2 and 36 (inclusive). If `ptr` is non-`NULL` it will contain a pointer to the end of the scan.

If the converted value falls out of range of corresponding return type, range error occurs (`errno` is set to `ERANGE`) and `ULONG_MAX` is returned. If no conversion can be performed, `0` is returned.

See also the Unix man page *strtoul(3)*.

Added in version 3.2.

long **PyOS_strtol**(const char *str, char **ptr, int base)

Część stabilnego ABI. Convert the initial part of the string in `str` to an `long` value according to the given `base`, which must be between `2` and `36` inclusive, or be the special value `0`.

Same as *PyOS_strtoul()*, but return a `long` value instead and `LONG_MAX` on overflows.

See also the Unix man page *strtol(3)*.

Added in version 3.2.

double **PyOS_string_to_double**(const char *s, char **endptr, *PyObject* *overflow_exception)

Część stabilnego ABI. Convert a string `s` to a `double`, raising a Python exception on failure. The set of accepted strings corresponds to the set of strings accepted by Python's `float()` constructor, except that `s` must not have leading or trailing whitespace. The conversion is independent of the current locale.

If `endptr` is `NULL`, convert the whole string. Raise `ValueError` and return `-1.0` if the string is not a valid representation of a floating-point number.

If endptr is not `NULL`, convert as much of the string as possible and set `*endptr` to point to the first unconverted character. If no initial segment of the string is the valid representation of a floating-point number, set `*endptr` to point to the beginning of the string, raise ValueError, and return `-1.0`.

If `s` represents a value that is too large to store in a float (for example, `"1e500"` is such a string on many platforms) then if `overflow_exception` is `NULL` return `Py_HUGE_VAL` (with an appropriate sign) and don't set any exception. Otherwise, `overflow_exception` must point to a Python exception object; raise that exception and return `-1.0`. In both cases, set `*endptr` to point to the first character after the converted value.

If any other error occurs during the conversion (for example an out-of-memory error), set the appropriate Python exception and return `-1.0`.

Added in version 3.1.

char ***PyOS_double_to_string**(double val, char format_code, int precision, int flags, int *ptype)

Część stabilnego ABI. Convert a `double` *val* to a string using supplied *format_code*, *precision*, and *flags*.

format_code must be one of `'e'`, `'E'`, `'f'`, `'F'`, `'g'`, `'G'` or `'r'`. For `'r'`, the supplied *precision* must be 0 and is ignored. The `'r'` format code specifies the standard `repr()` format.

flags can be zero or more of the values `Py_DTSF_SIGN`, `Py_DTSF_ADD_DOT_0`, or `Py_DTSF_ALT`, or-ed together:

- `Py_DTSF_SIGN` means to always precede the returned string with a sign character, even if *val* is non--negative.
- `Py_DTSF_ADD_DOT_0` means to ensure that the returned string will not look like an integer.
- `Py_DTSF_ALT` means to apply „alternate" formatting rules. See the documentation for the *PyOS_snprintf()* `'#'` specifier for details.

If *ptype* is non-`NULL`, then the value it points to will be set to one of `Py_DTST_FINITE`, `Py_DTST_INFINITE`, or `Py_DTST_NAN`, signifying that *val* is a finite number, an infinite number, or not a number, respectively.

The return value is a pointer to *buffer* with the converted string or `NULL` if the conversion failed. The caller is responsible for freeing the returned string by calling `PyMem_Free()`.

Added in version 3.1.

int **PyOS_stricmp**(const char *s1, const char *s2)

Case insensitive comparison of strings. The function works almost identically to `strcmp()` except that it ignores the case.

int **PyOS_strnicmp**(const char *s1, const char *s2, *Py_ssize_t* size)

Case insensitive comparison of strings. The function works almost identically to `strncmp()` except that it ignores the case.

6.8 PyHash API

See also the `PyTypeObject.tp_hash` member and numeric-hash.

type **Py_hash_t**

Hash value type: signed integer.

Added in version 3.2.

type **Py_uhash_t**

Hash value type: unsigned integer.

Added in version 3.2.

PyHASH_MODULUS

The Mersenne prime `P = 2**n -1`, used for numeric hash scheme.

Added in version 3.13.

PyHASH_BITS

The exponent `n` of `P` in `PyHASH_MODULUS`.

Added in version 3.13.

PyHASH_MULTIPLIER

Prime multiplier used in string and various other hashes.

Added in version 3.13.

PyHASH_INF

The hash value returned for a positive infinity.

Added in version 3.13.

PyHASH_IMAG

The multiplier used for the imaginary part of a complex number.

Added in version 3.13.

type **PyHash_FuncDef**

Hash function definition used by `PyHash_GetFuncDef()`.

const char ***name**

Hash function name (UTF-8 encoded string).

const int **hash_bits**

Internal size of the hash value in bits.

const int **seed_bits**

Size of seed input in bits.

Added in version 3.4.

PyHash_FuncDef ***PyHash_GetFuncDef**(void)

Get the hash function definition.

> **Zobacz także**
>
> **PEP 456** „Secure and interchangeable hash algorithm”.

Added in version 3.4.

Py_hash_t **Py_HashPointer**(const void *ptr)

Hash a pointer value: process the pointer value as an integer (cast it to `uintptr_t` internally). The pointer is not dereferenced.

The function cannot fail: it cannot return `-1`.

Added in version 3.13.

Py_hash_t **PyObject_GenericHash**(*PyObject* *obj)

Generic hashing function that is meant to be put into a type object's `tp_hash` slot. Its result only depends on the object's identity.

Szczegół implementacyjny CPythona: In CPython, it is equivalent to `Py_HashPointer()`.

Added in version 3.13.

6.9 Reflection

PyObject ***PyEval_GetBuiltins**(void)

Wartość zwracana: pożyczona referencja. Część stabilnego ABI. Niezalecane od wersji 3.13: Use `PyEval_GetFrameBuiltins()` instead.

Return a dictionary of the builtins in the current execution frame, or the interpreter of the thread state if no frame is currently executing.

PyObject ***PyEval_GetLocals**(void)

Wartość zwracana: pożyczona referencja. Część stabilnego ABI. Niezalecane od wersji 3.13: Use either `PyEval_GetFrameLocals()` to obtain the same behaviour as calling `locals()` in Python code, or else call `PyFrame_GetLocals()` on the result of `PyEval_GetFrame()` to access the `f_locals` attribute of the currently executing frame.

Return a mapping providing access to the local variables in the current execution frame, or `NULL` if no frame is currently executing.

Refer to `locals()` for details of the mapping returned at different scopes.

As this function returns a *borrowed reference*, the dictionary returned for *optimized scopes* is cached on the frame object and will remain alive as long as the frame object does. Unlike `PyEval_GetFrameLocals()` and `locals()`, subsequent calls to this function in the same frame will update the contents of the cached dictionary to reflect changes in the state of the local variables rather than returning a new snapshot.

Zmienione w wersji 3.13: As part of **PEP 667**, `PyFrame_GetLocals()`, `locals()`, and `FrameType.f_locals` no longer make use of the shared cache dictionary. Refer to the What's New entry for additional details.

PyObject ***PyEval_GetGlobals**(void)

Wartość zwracana: pożyczona referencja. Część stabilnego ABI. Niezalecane od wersji 3.13: Use `PyEval_GetFrameGlobals()` instead.

Return a dictionary of the global variables in the current execution frame, or `NULL` if no frame is currently executing.

PyFrameObject ***PyEval_GetFrame**(void)

Wartość zwracana: pożyczona referencja. Część stabilnego ABI. Return the current thread state's frame, which is `NULL` if no frame is currently executing.

See also `PyThreadState_GetFrame()`.

PyObject ***PyEval_GetFrameBuiltins**(void)

Wartość zwracana: nowa referencja. Część stabilnego ABI *od wersji 3.13.* Return a dictionary of the builtins in the current execution frame, or the interpreter of the thread state if no frame is currently executing.

Added in version 3.13.

PyObject ***PyEval_GetFrameLocals**(void)

Wartość zwracana: nowa referencja. Część stabilnego ABI *od wersji 3.13.* Return a dictionary of the local variables in the current execution frame, or `NULL` if no frame is currently executing. Equivalent to calling `locals()` in Python code.

To access `f_locals` on the current frame without making an independent snapshot in *optimized scopes*, call `PyFrame_GetLocals()` on the result of `PyEval_GetFrame()`.

Added in version 3.13.

PyObject ***PyEval_GetFrameGlobals**(void)

Wartość zwracana: nowa referencja. Część stabilnego ABI *od wersji 3.13.* Return a dictionary of the global variables in the current execution frame, or `NULL` if no frame is currently executing. Equivalent to calling `globals()` in Python code.

Added in version 3.13.

const char ***PyEval_GetFuncName**(*PyObject* *func)

Część stabilnego ABI. Return the name of *func* if it is a function, class or instance object, else the name of *func*s type.

const char ***PyEval_GetFuncDesc**(*PyObject* *func)

Część stabilnego ABI. Return a description string, depending on the type of *func*. Return values include „()" for functions and methods, „ constructor", „ instance", and „ object". Concatenated with the result of `PyEval_GetFuncName()`, the result will be a description of *func*.

6.10 Codec registry and support functions

int **PyCodec_Register**(*PyObject* *search_function)

Część stabilnego ABI. Register a new codec search function.

As side effect, this tries to load the `encodings` package, if not yet done, to make sure that it is always first in the list of search functions.

int **PyCodec_Unregister**(*PyObject* *search_function)

Część stabilnego ABI *od wersji 3.10.* Unregister a codec search function and clear the registry's cache. If the search function is not registered, do nothing. Return 0 on success. Raise an exception and return -1 on error.

Added in version 3.10.

int **PyCodec_KnownEncoding**(const char *encoding)

Część stabilnego ABI. Return `1` or `0` depending on whether there is a registered codec for the given *encoding*. This function always succeeds.

PyObject ***PyCodec_Encode**(*PyObject* *object, const char *encoding, const char *errors)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Generic codec based encoding API.

object is passed through the encoder function found for the given *encoding* using the error handling method defined by *errors*. *errors* may be `NULL` to use the default method defined for the codec. Raises a `LookupError` if no encoder can be found.

PyObject ***PyCodec_Decode**(*PyObject* *object, const char *encoding, const char *errors)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Generic codec based decoding API.

object is passed through the decoder function found for the given *encoding* using the error handling method defined by *errors*. *errors* may be `NULL` to use the default method defined for the codec. Raises a `LookupError` if no encoder can be found.

6.10.1 Codec lookup API

In the following functions, the *encoding* string is looked up converted to all lower-case characters, which makes encodings looked up through this mechanism effectively case-insensitive. If no codec is found, a `KeyError` is set and `NULL` returned.

PyObject ***PyCodec_Encoder**(const char *encoding)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Get an encoder function for the given *encoding*.

PyObject ***PyCodec_Decoder**(const char *encoding)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Get a decoder function for the given *encoding*.

PyObject ***PyCodec_IncrementalEncoder**(const char *encoding, const char *errors)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Get an `IncrementalEncoder` object for the given *encoding*.

PyObject ***PyCodec_IncrementalDecoder**(const char *encoding, const char *errors)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Get an `IncrementalDecoder` object for the given *encoding*.

PyObject ***PyCodec_StreamReader**(const char *encoding, *PyObject* *stream, const char *errors)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Get a `StreamReader` factory function for the given *encoding*.

PyObject ***PyCodec_StreamWriter**(const char *encoding, *PyObject* *stream, const char *errors)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Get a `StreamWriter` factory function for the given *encoding*.

6.10.2 Registry API for Unicode encoding error handlers

int **PyCodec_RegisterError**(const char *name, *PyObject* *error)

Część stabilnego ABI. Register the error handling callback function *error* under the given *name*. This callback function will be called by a codec when it encounters unencodable characters/undecodable bytes and *name* is specified as the error parameter in the call to the encode/decode function.

The callback gets a single argument, an instance of `UnicodeEncodeError`, `UnicodeDecodeError` or `UnicodeTranslateError` that holds information about the problematic sequence of characters or bytes and their offset in the original string (see *Unicode Exception Objects* for functions to extract this information). The callback must either raise the given exception, or return a two-item tuple containing the replacement for the problematic sequence, and an integer giving the offset in the original string at which encoding/decoding should be resumed.

Return `0` on success, `-1` on error.

PyObject ***PyCodec_LookupError**(const char *name)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Lookup the error handling callback function registered under *name*. As a special case `NULL` can be passed, in which case the error handling callback for „strict” will be returned.

PyObject ***PyCodec_StrictErrors**(*PyObject* *exc)

Wartość zwracana: zawsze NULL. Część stabilnego ABI. Raise *exc* as an exception.

PyObject ***PyCodec_IgnoreErrors**(*PyObject* *exc)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Ignore the unicode error, skipping the faulty input.

PyObject ***PyCodec_ReplaceErrors**(*PyObject* *exc)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Replace the unicode encode error with `?` or `U+FFFD`.

PyObject ***PyCodec_XMLCharRefReplaceErrors**(*PyObject* *exc)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Replace the unicode encode error with XML character references.

PyObject ***PyCodec_BackslashReplaceErrors**(*PyObject* *exc)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Replace the unicode encode error with backslash escapes (`\x`, `\u` and `\U`).

PyObject ***PyCodec_NameReplaceErrors**(*PyObject* *exc)

Wartość zwracana: nowa referencja. Część stabilnego ABI *od wersji 3.7*. Replace the unicode encode error with `\N{...}` escapes.

Added in version 3.5.

6.11 PyTime C API

Added in version 3.13.

The clock C API provides access to system clocks. It is similar to the Python `time` module.

For C API related to the `datetime` module, see *DateTime Objects*.

6.11.1 Typy

type **PyTime_t**

A timestamp or duration in nanoseconds, represented as a signed 64-bit integer.

The reference point for timestamps depends on the clock used. For example, `PyTime_Time()` returns timestamps relative to the UNIX epoch.

The supported range is around [-292.3 years; +292.3 years]. Using the Unix epoch (January 1st, 1970) as reference, the supported date range is around [1677-09-21; 2262-04-11]. The exact limits are exposed as constants:

PyTime_t **PyTime_MIN**

Minimum value of `PyTime_t`.

PyTime_t **PyTime_MAX**

Maximum value of `PyTime_t`.

6.11.2 Clock Functions

The following functions take a pointer to a `PyTime_t` that they set to the value of a particular clock. Details of each clock are given in the documentation of the corresponding Python function.

The functions return `0` on success, or `-1` (with an exception set) on failure.

On integer overflow, they set the `PyExc_OverflowError` exception and set `*result` to the value clamped to the `[PyTime_MIN; PyTime_MAX]` range. (On current systems, integer overflows are likely caused by misconfigured system time.)

As any other C API (unless otherwise specified), the functions must be called with the *GIL* held.

int **PyTime_Monotonic**(*PyTime_t* *result)

Read the monotonic clock. See `time.monotonic()` for important details on this clock.

int **PyTime_PerfCounter**(*PyTime_t* *result)

Read the performance counter. See `time.perf_counter()` for important details on this clock.

int **PyTime_Time**(*PyTime_t* *result)

Read the "wall clock" time. See `time.time()` for details important on this clock.

6.11.3 Raw Clock Functions

Similar to clock functions, but don't set an exception on error and don't require the caller to hold the GIL.

On success, the functions return `0`.

On failure, they set `*result` to `0` and return `-1`, *without* setting an exception. To get the cause of the error, acquire the GIL and call the regular (non-`Raw`) function. Note that the regular function may succeed after the `Raw` one failed.

int **`PyTime_MonotonicRaw`**(*PyTime_t* *result)

Similar to `PyTime_Monotonic()`, but don't set an exception on error and don't require holding the GIL.

int **`PyTime_PerfCounterRaw`**(*PyTime_t* *result)

Similar to `PyTime_PerfCounter()`, but don't set an exception on error and don't require holding the GIL.

int **`PyTime_TimeRaw`**(*PyTime_t* *result)

Similar to `PyTime_Time()`, but don't set an exception on error and don't require holding the GIL.

6.11.4 Conversion functions

double **`PyTime_AsSecondsDouble`**(*PyTime_t* t)

Convert a timestamp to a number of seconds as a C `double`.

The function cannot fail, but note that `double` has limited accuracy for large values.

6.12 Support for Perf Maps

On supported platforms (as of this writing, only Linux), the runtime can take advantage of *perf map files* to make Python functions visible to an external profiling tool (such as perf). A running process may create a file in the `/tmp` directory, which contains entries that can map a section of executable code to a name. This interface is described in the documentation of the Linux Perf tool.

In Python, these helper APIs can be used by libraries and features that rely on generating machine code on the fly.

Note that holding the Global Interpreter Lock (GIL) is not required for these APIs.

int **`PyUnstable_PerfMapState_Init`**(void)

> To jest *niestabilne API*. Może ulec zmianie bez ostrzeżenia w wydaniach minor.

Open the `/tmp/perf-$pid.map` file, unless it's already opened, and create a lock to ensure thread-safe writes to the file (provided the writes are done through `PyUnstable_WritePerfMapEntry()`). Normally, there's no need to call this explicitly; just use `PyUnstable_WritePerfMapEntry()` and it will initialize the state on first call.

Returns `0` on success, `-1` on failure to create/open the perf map file, or `-2` on failure to create a lock. Check `errno` for more information about the cause of a failure.

int **`PyUnstable_WritePerfMapEntry`**(const void *code_addr, unsigned int code_size, const char *entry_name)

> To jest *niestabilne API*. Może ulec zmianie bez ostrzeżenia w wydaniach minor.

Write one single entry to the `/tmp/perf-$pid.map` file. This function is thread safe. Here is what an example entry looks like:

```
# address       size  name
7f3529fcf759 b     py::bar:/run/t.py
```

Will call `PyUnstable_PerfMapState_Init()` before writing the entry, if the perf map file is not already opened. Returns `0` on success, or the same error codes as `PyUnstable_PerfMapState_Init()` on failure.

void **PyUnstable_PerfMapState_Fini**(void)

To jest *niestabilne API*. Może ulec zmianie bez ostrzeżenia w wydaniach minor.

Close the perf map file opened by `PyUnstable_PerfMapState_Init()`. This is called by the runtime itself during interpreter shut-down. In general, there shouldn't be a reason to explicitly call this, except to handle specific scenarios such as forking.

ROZDZIAŁ 7

Warstwa obiektów abstrakcyjnych

Funkcje, których dotyczy ten rozdział działają na obiektach Pythona bez względu na ich typ oraz na wielu klasach typów obiektów (np. wszystkie typy numeryczne oraz sekwencyjne). Użyte na nieobsługiwanych typach obiektów rzucą wyjątek.

Nie da się używać tych funckcji na obiektach, które nie zostały prawidłowo zainicjowane jak np. lista utworzona za pomocą `PyList_New()`, której elementom nie nadano jeszcze wartości innej niż `NULL`.

7.1 Object Protocol

PyObject ***Py_GetConstant**(unsigned int constant_id)

Część stabilnego ABI *od wersji 3.13*. Get a *strong reference* to a constant.

Set an exception and return `NULL` if *constant_id* is invalid.

constant_id must be one of these constant identifiers:

Constant Identifier	Wartość	Returned object
Py_CONSTANT_NONE	0	None
Py_CONSTANT_FALSE	1	False
Py_CONSTANT_TRUE	2	True
Py_CONSTANT_ELLIPSIS	3	Ellipsis
Py_CONSTANT_NOT_IMPLEMENTI	4	NotImplemented
Py_CONSTANT_ZERO	5	0
Py_CONSTANT_ONE	6	1
Py_CONSTANT_EMPTY_STR	7	''
Py_CONSTANT_EMPTY_BYTES	8	b''
Py_CONSTANT_EMPTY_TUPLE	9	()

Numeric values are only given for projects which cannot use the constant identifiers.

Added in version 3.13.

Szczegół implementacyjny CPythona: In CPython, all of these constants are *immortal*.

PyObject ***Py_GetConstantBorrowed**(unsigned int constant_id)

Część stabilnego ABI *od wersji 3.13.* Similar to `Py_GetConstant()`, but return a *borrowed reference*.

This function is primarily intended for backwards compatibility: using `Py_GetConstant()` is recommended for new code.

The reference is borrowed from the interpreter, and is valid until the interpreter finalization.

Added in version 3.13.

PyObject ***Py_NotImplemented**

The `NotImplemented` singleton, used to signal that an operation is not implemented for the given type combination.

Py_RETURN_NOTIMPLEMENTED

Properly handle returning `Py_NotImplemented` from within a C function (that is, create a new *strong reference* to `NotImplemented` and return it).

Py_PRINT_RAW

Flag to be used with multiple functions that print the object (like `PyObject_Print()` and `PyFile_WriteObject()`). If passed, these function would use the `str()` of the object instead of the `repr()`.

int **PyObject_Print**(*PyObject* *o, FILE *fp, int flags)

Print an object *o*, on file *fp*. Returns `-1` on error. The flags argument is used to enable certain printing options. The only option currently supported is `Py_PRINT_RAW`; if given, the `str()` of the object is written instead of the `repr()`.

int **PyObject_HasAttrWithError**(*PyObject* *o, const char *attr_name)

Część stabilnego ABI *od wersji 3.13*. Returns `1` if *o* has the attribute *attr_name*, and `0` otherwise. This is equivalent to the Python expression `hasattr(o, attr_name)`. On failure, return `-1`.

Added in version 3.13.

int **PyObject_HasAttrStringWithError**(*PyObject* *o, const char *attr_name)

Część stabilnego ABI *od wersji 3.13*. This is the same as `PyObject_HasAttrWithError()`, but *attr_name* is specified as a `const char*` UTF-8 encoded bytes string, rather than a `PyObject*`.

Added in version 3.13.

int **PyObject_HasAttr**(*PyObject* *o, *PyObject* *attr_name)

Część stabilnego ABI. Returns `1` if *o* has the attribute *attr_name*, and `0` otherwise. This function always succeeds.

> **Informacja**
>
> Exceptions that occur when this calls `__getattr__()` and `__getattribute__()` methods are silently ignored. For proper error handling, use `PyObject_HasAttrWithError()`, `PyObject_GetOptionalAttr()` or `PyObject_GetAttr()` instead.

int **PyObject_HasAttrString**(*PyObject* *o, const char *attr_name)

Część stabilnego ABI. This is the same as `PyObject_HasAttr()`, but *attr_name* is specified as a `const char*` UTF-8 encoded bytes string, rather than a `PyObject*`.

> **Informacja**
>
> Exceptions that occur when this calls `__getattr__()` and `__getattribute__()` methods or while creating the temporary `str` object are silently ignored. For proper error handling, use `PyObject_HasAttrStringWithError()`, `PyObject_GetOptionalAttrString()` or `PyObject_GetAttrString()` instead.

PyObject ***PyObject_GetAttr**(*PyObject* *o, *PyObject* *attr_name)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Retrieve an attribute named *attr_name* from object *o*. Returns the attribute value on success, or `NULL` on failure. This is the equivalent of the Python expression `o.attr_name`.

If the missing attribute should not be treated as a failure, you can use `PyObject_GetOptionalAttr()` instead.

PyObject ***PyObject_GetAttrString**(*PyObject* *o, const char *attr_name)

Wartość zwracana: nowa referencja. Część stabilnego ABI. This is the same as `PyObject_GetAttr()`, but *attr_name* is specified as a `const char*` UTF-8 encoded bytes string, rather than a `PyObject*`.

If the missing attribute should not be treated as a failure, you can use `PyObject_GetOptionalAttrString()` instead.

int **PyObject_GetOptionalAttr**(*PyObject* *obj, *PyObject* *attr_name, *PyObject* **result);

Część stabilnego ABI *od wersji 3.13*. Variant of `PyObject_GetAttr()` which doesn't raise `AttributeError` if the attribute is not found.

If the attribute is found, return `1` and set **result* to a new *strong reference* to the attribute. If the attribute is not found, return `0` and set **result* to `NULL`; the `AttributeError` is silenced. If an error other than `AttributeError` is raised, return `-1` and set **result* to `NULL`.

Added in version 3.13.

int **PyObject_GetOptionalAttrString**(*PyObject* *obj, const char *attr_name, *PyObject* **result);

Część stabilnego ABI *od wersji 3.13.* This is the same as `PyObject_GetOptionalAttr()`, but *attr_name* is specified as a `const char*` UTF-8 encoded bytes string, rather than a `PyObject*`.

Added in version 3.13.

PyObject ***PyObject_GenericGetAttr**(*PyObject* *o, *PyObject* *name)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Generic attribute getter function that is meant to be put into a type object's `tp_getattro` slot. It looks for a descriptor in the dictionary of classes in the object's MRO as well as an attribute in the object's `__dict__` (if present). As outlined in descriptors, data descriptors take preference over instance attributes, while non-data descriptors don't. Otherwise, an `AttributeError` is raised.

int **PyObject_SetAttr**(*PyObject* *o, *PyObject* *attr_name, *PyObject* *v)

Część stabilnego ABI. Set the value of the attribute named *attr_name*, for object *o*, to the value *v*. Raise an exception and return `-1` on failure; return `0` on success. This is the equivalent of the Python statement `o.attr_name = v`.

If *v* is `NULL`, the attribute is deleted. This behaviour is deprecated in favour of using `PyObject_DelAttr()`, but there are currently no plans to remove it.

int **PyObject_SetAttrString**(*PyObject* *o, const char *attr_name, *PyObject* *v)

Część stabilnego ABI. This is the same as `PyObject_SetAttr()`, but *attr_name* is specified as a `const char*` UTF-8 encoded bytes string, rather than a `PyObject*`.

If *v* is `NULL`, the attribute is deleted, but this feature is deprecated in favour of using `PyObject_DelAttrString()`.

The number of different attribute names passed to this function should be kept small, usually by using a statically allocated string as *attr_name*. For attribute names that aren't known at compile time, prefer calling `PyUnicode_FromString()` and `PyObject_SetAttr()` directly. For more details, see `PyUnicode_InternFromString()`, which may be used internally to create a key object.

int **PyObject_GenericSetAttr**(*PyObject* *o, *PyObject* *name, *PyObject* *value)

Część stabilnego ABI. Generic attribute setter and deleter function that is meant to be put into a type object's `tp_setattro` slot. It looks for a data descriptor in the dictionary of classes in the object's MRO, and if found it takes preference over setting or deleting the attribute in the instance dictionary. Otherwise, the attribute is set or deleted in the object's `__dict__` (if present). On success, `0` is returned, otherwise an `AttributeError` is raised and `-1` is returned.

int **PyObject_DelAttr**(*PyObject* *o, *PyObject* *attr_name)

Część stabilnego ABI *od wersji 3.13.* Delete attribute named *attr_name*, for object *o*. Returns `-1` on failure. This is the equivalent of the Python statement `del o.attr_name`.

int **PyObject_DelAttrString**(*PyObject* *o, const char *attr_name)

Część stabilnego ABI *od wersji 3.13.* This is the same as `PyObject_DelAttr()`, but *attr_name* is specified as a `const char*` UTF-8 encoded bytes string, rather than a `PyObject*`.

The number of different attribute names passed to this function should be kept small, usually by using a statically allocated string as *attr_name*. For attribute names that aren't known at compile time, prefer calling `PyUnicode_FromString()` and `PyObject_DelAttr()` directly. For more details, see `PyUnicode_InternFromString()`, which may be used internally to create a key object for lookup.

PyObject ***PyObject_GenericGetDict**(*PyObject* *o, void *context)

Wartość zwracana: nowa referencja. Część stabilnego ABI *od wersji 3.10.* A generic implementation for the getter of a `__dict__` descriptor. It creates the dictionary if necessary.

This function may also be called to get the `__dict__` of the object *o*. Pass `NULL` for *context* when calling it. Since this function may need to allocate memory for the dictionary, it may be more efficient to call `PyObject_GetAttr()` when accessing an attribute on the object.

On failure, returns `NULL` with an exception set.

Added in version 3.3.

int **PyObject_GenericSetDict**(*PyObject* *o, *PyObject* *value, void *context)

Część stabilnego ABI *od wersji 3.7.* A generic implementation for the setter of a `__dict__` descriptor. This implementation does not allow the dictionary to be deleted.

Added in version 3.3.

PyObject ****_PyObject_GetDictPtr**(*PyObject* *obj)

Return a pointer to `__dict__` of the object *obj*. If there is no `__dict__`, return `NULL` without setting an exception.

This function may need to allocate memory for the dictionary, so it may be more efficient to call `PyObject_GetAttr()` when accessing an attribute on the object.

PyObject ***PyObject_RichCompare**(*PyObject* *o1, *PyObject* *o2, int opid)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Compare the values of *o1* and *o2* using the operation specified by *opid*, which must be one of `Py_LT`, `Py_LE`, `Py_EQ`, `Py_NE`, `Py_GT`, or `Py_GE`, corresponding to `<`, `<=`, `==`, `!=`, `>`, or `>=` respectively. This is the equivalent of the Python expression `o1 op o2`, where `op` is the operator corresponding to *opid*. Returns the value of the comparison on success, or `NULL` on failure.

int **PyObject_RichCompareBool**(*PyObject* *o1, *PyObject* *o2, int opid)

Część stabilnego ABI. Compare the values of *o1* and *o2* using the operation specified by *opid*, like `PyObject_RichCompare()`, but returns `-1` on error, `0` if the result is false, `1` otherwise.

Informacja

If *o1* and *o2* are the same object, `PyObject_RichCompareBool()` will always return `1` for `Py_EQ` and `0` for `Py_NE`.

PyObject ***PyObject_Format**(*PyObject* *obj, *PyObject* *format_spec)

Część stabilnego ABI. Format *obj* using *format_spec*. This is equivalent to the Python expression `format(obj, format_spec)`.

format_spec may be `NULL`. In this case the call is equivalent to `format(obj)`. Returns the formatted string on success, `NULL` on failure.

PyObject ***PyObject_Repr**(*PyObject* *o)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Compute a string representation of object *o*. Returns the string representation on success, `NULL` on failure. This is the equivalent of the Python expression `repr(o)`. Called by the `repr()` built-in function.

Zmienione w wersji 3.4: This function now includes a debug assertion to help ensure that it does not silently discard an active exception.

PyObject ***PyObject_ASCII**(*PyObject* *o)

Wartość zwracana: nowa referencja. Część stabilnego ABI. As `PyObject_Repr()`, compute a string representation of object *o*, but escape the non-ASCII characters in the string returned by `PyObject_Repr()` with `\x`, `\u` or `\U` escapes. This generates a string similar to that returned by `PyObject_Repr()` in Python 2. Called by the `ascii()` built-in function.

PyObject ***PyObject_Str**(*PyObject* *o)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Compute a string representation of object *o*. Returns the string representation on success, `NULL` on failure. This is the equivalent of the Python expression `str(o)`. Called by the `str()` built-in function and, therefore, by the `print()` function.

Zmienione w wersji 3.4: This function now includes a debug assertion to help ensure that it does not silently discard an active exception.

PyObject ***PyObject_Bytes*** (*PyObject* *o)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Compute a bytes representation of object *o*. `NULL` is returned on failure and a bytes object on success. This is equivalent to the Python expression `bytes(o)`, when *o* is not an integer. Unlike `bytes(o)`, a TypeError is raised when *o* is an integer instead of a zero-initialized bytes object.

int ***PyObject_IsSubclass*** (*PyObject* *derived, *PyObject* *cls)

Część stabilnego ABI. Return `1` if the class *derived* is identical to or derived from the class *cls*, otherwise return `0`. In case of an error, return `-1`.

If *cls* is a tuple, the check will be done against every entry in *cls*. The result will be `1` when at least one of the checks returns `1`, otherwise it will be `0`.

If *cls* has a `__subclasscheck__()` method, it will be called to determine the subclass status as described in **PEP 3119**. Otherwise, *derived* is a subclass of *cls* if it is a direct or indirect subclass, i.e. contained in `cls.__mro__`.

Normally only class objects, i.e. instances of `type` or a derived class, are considered classes. However, objects can override this by having a `__bases__` attribute (which must be a tuple of base classes).

int ***PyObject_IsInstance*** (*PyObject* *inst, *PyObject* *cls)

Część stabilnego ABI. Return `1` if *inst* is an instance of the class *cls* or a subclass of *cls*, or `0` if not. On error, returns `-1` and sets an exception.

If *cls* is a tuple, the check will be done against every entry in *cls*. The result will be `1` when at least one of the checks returns `1`, otherwise it will be `0`.

If *cls* has a `__instancecheck__()` method, it will be called to determine the subclass status as described in **PEP 3119**. Otherwise, *inst* is an instance of *cls* if its class is a subclass of *cls*.

An instance *inst* can override what is considered its class by having a `__class__` attribute.

An object *cls* can override if it is considered a class, and what its base classes are, by having a `__bases__` attribute (which must be a tuple of base classes).

Py_hash_t ***PyObject_Hash*** (*PyObject* *o)

Część stabilnego ABI. Compute and return the hash value of an object *o*. On failure, return `-1`. This is the equivalent of the Python expression `hash(o)`.

Zmienione w wersji 3.2: The return type is now Py_hash_t. This is a signed integer the same size as `Py_ssize_t`.

Py_hash_t ***PyObject_HashNotImplemented*** (*PyObject* *o)

Część stabilnego ABI. Set a `TypeError` indicating that `type(o)` is not *hashable* and return `-1`. This function receives special treatment when stored in a `tp_hash` slot, allowing a type to explicitly indicate to the interpreter that it is not hashable.

int ***PyObject_IsTrue*** (*PyObject* *o)

Część stabilnego ABI. Returns `1` if the object *o* is considered to be true, and `0` otherwise. This is equivalent to the Python expression `not not o`. On failure, return `-1`.

int ***PyObject_Not*** (*PyObject* *o)

Część stabilnego ABI. Returns `0` if the object *o* is considered to be true, and `1` otherwise. This is equivalent to the Python expression `not o`. On failure, return `-1`.

PyObject ***PyObject_Type*** (*PyObject* *o)

Wartość zwracana: nowa referencja. Część stabilnego ABI. When *o* is non-`NULL`, returns a type object corresponding to the object type of object *o*. On failure, raises `SystemError` and returns `NULL`. This is equivalent to the Python expression `type(o)`. This function creates a new *strong reference* to the return value. There's really no reason to use this function instead of the `Py_TYPE()` function, which returns a pointer of type `PyTypeObject*`, except when a new *strong reference* is needed.

int **PyObject_TypeCheck**(*PyObject* *o, *PyTypeObject* *type)

Return non-zero if the object *o* is of type *type* or a subtype of *type*, and `0` otherwise. Both parameters must be non-`NULL`.

Py_ssize_t **PyObject_Size**(*PyObject* *o)

Py_ssize_t **PyObject_Length**(*PyObject* *o)

Część stabilnego ABI. Return the length of object *o*. If the object *o* provides either the sequence and mapping protocols, the sequence length is returned. On error, `-1` is returned. This is the equivalent to the Python expression `len(o)`.

Py_ssize_t **PyObject_LengthHint**(*PyObject* *o, *Py_ssize_t* defaultvalue)

Return an estimated length for the object *o*. First try to return its actual length, then an estimate using `__length_hint__()`, and finally return the default value. On error return `-1`. This is the equivalent to the Python expression `operator.length_hint(o, defaultvalue)`.

Added in version 3.4.

PyObject ***PyObject_GetItem**(*PyObject* *o, *PyObject* *key)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return element of *o* corresponding to the object *key* or `NULL` on failure. This is the equivalent of the Python expression `o[key]`.

int **PyObject_SetItem**(*PyObject* *o, *PyObject* *key, *PyObject* *v)

Część stabilnego ABI. Map the object *key* to the value *v*. Raise an exception and return `-1` on failure; return `0` on success. This is the equivalent of the Python statement `o[key] = v`. This function *does not* steal a reference to *v*.

int **PyObject_DelItem**(*PyObject* *o, *PyObject* *key)

Część stabilnego ABI. Remove the mapping for the object *key* from the object *o*. Return `-1` on failure. This is equivalent to the Python statement `del o[key]`.

PyObject ***PyObject_Dir**(*PyObject* *o)

Wartość zwracana: nowa referencja. Część stabilnego ABI. This is equivalent to the Python expression `dir(o)`, returning a (possibly empty) list of strings appropriate for the object argument, or `NULL` if there was an error. If the argument is `NULL`, this is like the Python `dir()`, returning the names of the current locals; in this case, if no execution frame is active then `NULL` is returned but `PyErr_Occurred()` will return false.

PyObject ***PyObject_GetIter**(*PyObject* *o)

Wartość zwracana: nowa referencja. Część stabilnego ABI. This is equivalent to the Python expression `iter(o)`. It returns a new iterator for the object argument, or the object itself if the object is already an iterator. Raises `TypeError` and returns `NULL` if the object cannot be iterated.

PyObject ***PyObject_GetAIter**(*PyObject* *o)

Wartość zwracana: nowa referencja. Część stabilnego ABI *od wersji 3.10.* This is the equivalent to the Python expression `aiter(o)`. Takes an `AsyncIterable` object and returns an `AsyncIterator` for it. This is typically a new iterator but if the argument is an `AsyncIterator`, this returns itself. Raises `TypeError` and returns `NULL` if the object cannot be iterated.

Added in version 3.10.

void ***PyObject_GetTypeData**(*PyObject* *o, *PyTypeObject* *cls)

Część stabilnego ABI *od wersji 3.12.* Get a pointer to subclass-specific data reserved for *cls*.

The object *o* must be an instance of *cls*, and *cls* must have been created using negative `PyType_Spec.basicsize`. Python does not check this.

On error, set an exception and return `NULL`.

Added in version 3.12.

Py_ssize_t **PyType_GetTypeDataSize**(*PyTypeObject* *cls)

Część stabilnego ABI *od wersji 3.12.* Return the size of the instance memory space reserved for *cls*, i.e. the size of the memory `PyObject_GetTypeData()` returns.

This may be larger than requested using `-PyType_Spec.basicsize`; it is safe to use this larger size (e.g. with `memset()`).

The type *cls* **must** have been created using negative `PyType_Spec.basicsize`. Python does not check this.

On error, set an exception and return a negative value.

Added in version 3.12.

void ***PyObject_GetItemData**(*PyObject* *o)

Get a pointer to per-item data for a class with `Py_TPFLAGS_ITEMS_AT_END`.

On error, set an exception and return `NULL`. `TypeError` is raised if *o* does not have `Py_TPFLAGS_ITEMS_AT_END` set.

Added in version 3.12.

int **PyObject_VisitManagedDict**(*PyObject* *obj, *visitproc* visit, void *arg)

Visit the managed dictionary of *obj*.

This function must only be called in a traverse function of the type which has the `Py_TPFLAGS_MANAGED_DICT` flag set.

Added in version 3.13.

void **PyObject_ClearManagedDict**(*PyObject* *obj)

Clear the managed dictionary of *obj*.

This function must only be called in a traverse function of the type which has the `Py_TPFLAGS_MANAGED_DICT` flag set.

Added in version 3.13.

7.2 Call Protocol

CPython supports two different calling protocols: *tp_call* and vectorcall.

7.2.1 The *tp_call* Protocol

Instances of classes that set `tp_call` are callable. The signature of the slot is:

```
PyObject *tp_call(PyObject *callable, PyObject *args, PyObject *kwargs);
```

A call is made using a tuple for the positional arguments and a dict for the keyword arguments, similarly to `callable(*args, **kwargs)` in Python code. *args* must be non-NULL (use an empty tuple if there are no arguments) but *kwargs* may be *NULL* if there are no keyword arguments.

This convention is not only used by *tp_call*: `tp_new` and `tp_init` also pass arguments this way.

To call an object, use `PyObject_Call()` or another *call API*.

7.2.2 The Vectorcall Protocol

Added in version 3.9.

The vectorcall protocol was introduced in **PEP 590** as an additional protocol for making calls more efficient.

As rule of thumb, CPython will prefer the vectorcall for internal calls if the callable supports it. However, this is not a hard rule. Additionally, some third-party extensions use *tp_call* directly (rather than using `PyObject_Call()`). Therefore, a class supporting vectorcall must also implement `tp_call`. Moreover, the callable must behave the same regardless of which protocol is used. The recommended way to achieve this is by setting `tp_call` to `PyVectorcall_Call()`. This bears repeating:

Ostrzeżenie

A class supporting vectorcall **must** also implement `tp_call` with the same semantics.

Zmienione w wersji 3.12: The `Py_TPFLAGS_HAVE_VECTORCALL` flag is now removed from a class when the class's `__call__()` method is reassigned. (This internally sets `tp_call` only, and thus may make it behave differently than the vectorcall function.) In earlier Python versions, vectorcall should only be used with `immutable` or static types.

A class should not implement vectorcall if that would be slower than *tp_call*. For example, if the callee needs to convert the arguments to an args tuple and kwargs dict anyway, then there is no point in implementing vectorcall.

Classes can implement the vectorcall protocol by enabling the `Py_TPFLAGS_HAVE_VECTORCALL` flag and setting `tp_vectorcall_offset` to the offset inside the object structure where a *vectorcallfunc* appears. This is a pointer to a function with the following signature:

typedef *PyObject* *(***vectorcallfunc**)(*PyObject* *callable, *PyObject* *const *args, size_t nargsf, *PyObject* *kwnames)

Część stabilnego ABI *od wersji 3.12.*

- *callable* is the object being called.
- ***args* is a C array consisting of the positional arguments followed by the**
  values of the keyword arguments. This can be *NULL* if there are no arguments.
- ***nargsf* is the number of positional arguments plus possibly the**
  `PY_VECTORCALL_ARGUMENTS_OFFSET` flag. To get the actual number of positional arguments from *nargsf*, use `PyVectorcall_NARGS()`.
- ***kwnames* is a tuple containing the names of the keyword arguments;**
  in other words, the keys of the kwargs dict. These names must be strings (instances of `str` or a subclass) and they must be unique. If there are no keyword arguments, then *kwnames* can instead be *NULL*.

PY_VECTORCALL_ARGUMENTS_OFFSET

Część stabilnego ABI *od wersji 3.12.* If this flag is set in a vectorcall *nargsf* argument, the callee is allowed to temporarily change `args[-1]`. In other words, *args* points to argument 1 (not 0) in the allocated vector. The callee must restore the value of `args[-1]` before returning.

For `PyObject_VectorcallMethod()`, this flag means instead that `args[0]` may be changed.

Whenever they can do so cheaply (without additional allocation), callers are encouraged to use `PY_VECTORCALL_ARGUMENTS_OFFSET`. Doing so will allow callables such as bound methods to make their onward calls (which include a prepended *self* argument) very efficiently.

Added in version 3.8.

To call an object that implements vectorcall, use a *call API* function as with any other callable. `PyObject_Vectorcall()` will usually be most efficient.

Kontrola Rekursji

When using *tp_call*, callees do not need to worry about *recursion*: CPython uses `Py_EnterRecursiveCall()` and `Py_LeaveRecursiveCall()` for calls made using *tp_call*.

For efficiency, this is not the case for calls done using vectorcall: the callee should use *Py_EnterRecursiveCall* and *Py_LeaveRecursiveCall* if needed.

Vectorcall Support API

Py_ssize_t **PyVectorcall_NARGS**(size_t nargsf)

Część stabilnego ABI *od wersji 3.12.* Given a vectorcall *nargsf* argument, return the actual number of arguments. Currently equivalent to:

```
(Py_ssize_t)(nargsf & ~PY_VECTORCALL_ARGUMENTS_OFFSET)
```

However, the function `PyVectorcall_NARGS` should be used to allow for future extensions.

Added in version 3.8.

vectorcallfunc **PyVectorcall_Function**(*PyObject* *op)

If *op* does not support the vectorcall protocol (either because the type does not or because the specific instance does not), return *NULL*. Otherwise, return the vectorcall function pointer stored in *op*. This function never raises an exception.

This is mostly useful to check whether or not *op* supports vectorcall, which can be done by checking `PyVectorcall_Function(op) != NULL`.

Added in version 3.9.

PyObject ***PyVectorcall_Call**(*PyObject* *callable, *PyObject* *tuple, *PyObject* *dict)

Część stabilnego ABI *od wersji 3.12*. Call *callable*'s `vectorcallfunc` with positional and keyword arguments given in a tuple and dict, respectively.

This is a specialized function, intended to be put in the `tp_call` slot or be used in an implementation of `tp_call`. It does not check the `Py_TPFLAGS_HAVE_VECTORCALL` flag and it does not fall back to `tp_call`.

Added in version 3.8.

7.2.3 Object Calling API

Various functions are available for calling a Python object. Each converts its arguments to a convention supported by the called object – either *tp_call* or vectorcall. In order to do as little conversion as possible, pick one that best fits the format of data you have available.

The following table summarizes the available functions; please see individual documentation for details.

Function	callable	args	kwargs
`PyObject_Call()`	`PyObject *`	tuple	dict/`NULL`
`PyObject_CallNoArgs()`	`PyObject *`	—	—
`PyObject_CallOneArg()`	`PyObject *`	1 object	—
`PyObject_CallObject()`	`PyObject *`	tuple/`NULL`	—
`PyObject_CallFunction()`	`PyObject *`	format	—
`PyObject_CallMethod()`	obj + `char*`	format	—
`PyObject_CallFunctionObjArgs()`	`PyObject *`	variadic	—
`PyObject_CallMethodObjArgs()`	obj + name	variadic	—
`PyObject_CallMethodNoArgs()`	obj + name	—	—
`PyObject_CallMethodOneArg()`	obj + name	1 object	—
`PyObject_Vectorcall()`	`PyObject *`	vectorcall	vectorcall
`PyObject_VectorcallDict()`	`PyObject *`	vectorcall	dict/`NULL`
`PyObject_VectorcallMethod()`	arg + name	vectorcall	vectorcall

PyObject ***PyObject_Call**(*PyObject* *callable, *PyObject* *args, *PyObject* *kwargs)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Call a callable Python object *callable*, with arguments given by the tuple *args*, and named arguments given by the dictionary *kwargs*.

args must not be *NULL*; use an empty tuple if no arguments are needed. If no named arguments are needed, *kwargs* can be *NULL*.

Return the result of the call on success, or raise an exception and return *NULL* on failure.

This is the equivalent of the Python expression: `callable(*args, **kwargs)`.

PyObject ***PyObject_CallNoArgs*** (*PyObject* *callable)

Wartość zwracana: nowa referencja. Część stabilnego ABI *od wersji 3.10.* Call a callable Python object *callable* without any arguments. It is the most efficient way to call a callable Python object without any argument.

Return the result of the call on success, or raise an exception and return *NULL* on failure.

Added in version 3.9.

PyObject ***PyObject_CallOneArg*** (*PyObject* *callable, *PyObject* *arg)

Wartość zwracana: nowa referencja. Call a callable Python object *callable* with exactly 1 positional argument *arg* and no keyword arguments.

Return the result of the call on success, or raise an exception and return *NULL* on failure.

Added in version 3.9.

PyObject ***PyObject_CallObject*** (*PyObject* *callable, *PyObject* *args)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Call a callable Python object *callable*, with arguments given by the tuple *args*. If no arguments are needed, then *args* can be *NULL*.

Return the result of the call on success, or raise an exception and return *NULL* on failure.

This is the equivalent of the Python expression: `callable(*args)`.

PyObject ***PyObject_CallFunction*** (*PyObject* *callable, const char *format, ...)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Call a callable Python object *callable*, with a variable number of C arguments. The C arguments are described using a `Py_BuildValue()` style format string. The format can be *NULL*, indicating that no arguments are provided.

Return the result of the call on success, or raise an exception and return *NULL* on failure.

This is the equivalent of the Python expression: `callable(*args)`.

Note that if you only pass `PyObject*` args, `PyObject_CallFunctionObjArgs()` is a faster alternative.

Zmienione w wersji 3.4: The type of *format* was changed from `char *`.

PyObject ***PyObject_CallMethod*** (*PyObject* *obj, const char *name, const char *format, ...)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Call the method named *name* of object *obj* with a variable number of C arguments. The C arguments are described by a `Py_BuildValue()` format string that should produce a tuple.

The format can be *NULL*, indicating that no arguments are provided.

Return the result of the call on success, or raise an exception and return *NULL* on failure.

This is the equivalent of the Python expression: `obj.name(arg1, arg2, ...)`.

Note that if you only pass `PyObject*` args, `PyObject_CallMethodObjArgs()` is a faster alternative.

Zmienione w wersji 3.4: The types of *name* and *format* were changed from `char *`.

PyObject ***PyObject_CallFunctionObjArgs*** (*PyObject* *callable, ...)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Call a callable Python object *callable*, with a variable number of `PyObject*` arguments. The arguments are provided as a variable number of parameters followed by *NULL*.

Return the result of the call on success, or raise an exception and return *NULL* on failure.

This is the equivalent of the Python expression: `callable(arg1, arg2, ...)`.

PyObject ***PyObject_CallMethodObjArgs*** (*PyObject* *obj, *PyObject* *name, ...)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Call a method of the Python object *obj*, where the name of the method is given as a Python string object in *name*. It is called with a variable number of `PyObject*` arguments. The arguments are provided as a variable number of parameters followed by *NULL*.

Return the result of the call on success, or raise an exception and return *NULL* on failure.

PyObject ***PyObject_CallMethodNoArgs*** (*PyObject* *obj, *PyObject* *name)

Call a method of the Python object *obj* without arguments, where the name of the method is given as a Python string object in *name*.

Return the result of the call on success, or raise an exception and return *NULL* on failure.

Added in version 3.9.

PyObject ***PyObject_CallMethodOneArg*** (*PyObject* *obj, *PyObject* *name, *PyObject* *arg)

Call a method of the Python object *obj* with a single positional argument *arg*, where the name of the method is given as a Python string object in *name*.

Return the result of the call on success, or raise an exception and return *NULL* on failure.

Added in version 3.9.

PyObject ***PyObject_Vectorcall*** (*PyObject* *callable, *PyObject* *const *args, size_t nargsf, *PyObject* *kwnames)

Część stabilnego ABI *od wersji 3.12.* Call a callable Python object *callable*. The arguments are the same as for `vectorcallfunc`. If *callable* supports *vectorcall*, this directly calls the vectorcall function stored in *callable*.

Return the result of the call on success, or raise an exception and return *NULL* on failure.

Added in version 3.9.

PyObject ***PyObject_VectorcallDict*** (*PyObject* *callable, *PyObject* *const *args, size_t nargsf, *PyObject* *kwdict)

Call *callable* with positional arguments passed exactly as in the *vectorcall* protocol, but with keyword arguments passed as a dictionary *kwdict*. The *args* array contains only the positional arguments.

Regardless of which protocol is used internally, a conversion of arguments needs to be done. Therefore, this function should only be used if the caller already has a dictionary ready to use for the keyword arguments, but not a tuple for the positional arguments.

Added in version 3.9.

PyObject ***PyObject_VectorcallMethod*** (*PyObject* *name, *PyObject* *const *args, size_t nargsf, *PyObject* *kwnames)

Część stabilnego ABI *od wersji 3.12.* Call a method using the vectorcall calling convention. The name of the method is given as a Python string *name*. The object whose method is called is *args[0]*, and the *args* array starting at *args[1]* represents the arguments of the call. There must be at least one positional argument. *nargsf* is the number of positional arguments including *args[0]*, plus `PY_VECTORCALL_ARGUMENTS_OFFSET` if the value of `args[0]` may temporarily be changed. Keyword arguments can be passed just like in `PyObject_Vectorcall()`.

If the object has the `Py_TPFLAGS_METHOD_DESCRIPTOR` feature, this will call the unbound method object with the full *args* vector as arguments.

Return the result of the call on success, or raise an exception and return *NULL* on failure.

Added in version 3.9.

7.2.4 Call Support API

int ***PyCallable_Check*** (*PyObject* *o)

Część stabilnego ABI. Determine if the object *o* is callable. Return `1` if the object is callable and `0` otherwise. This function always succeeds.

7.3 Number Protocol

int **PyNumber_Check**(*PyObject* *o)

Część stabilnego ABI. Returns `1` if the object *o* provides numeric protocols, and false otherwise. This function always succeeds.

Zmienione w wersji 3.8: Returns `1` if *o* is an index integer.

PyObject ***PyNumber_Add**(*PyObject* *o1, *PyObject* *o2)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Returns the result of adding *o1* and *o2*, or `NULL` on failure. This is the equivalent of the Python expression `o1 + o2`.

PyObject ***PyNumber_Subtract**(*PyObject* *o1, *PyObject* *o2)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Returns the result of subtracting *o2* from *o1*, or `NULL` on failure. This is the equivalent of the Python expression `o1 - o2`.

PyObject ***PyNumber_Multiply**(*PyObject* *o1, *PyObject* *o2)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Returns the result of multiplying *o1* and *o2*, or `NULL` on failure. This is the equivalent of the Python expression `o1 * o2`.

PyObject ***PyNumber_MatrixMultiply**(*PyObject* *o1, *PyObject* *o2)

Wartość zwracana: nowa referencja. Część stabilnego ABI *od wersji 3.7.* Returns the result of matrix multiplication on *o1* and *o2*, or `NULL` on failure. This is the equivalent of the Python expression `o1 @ o2`.

Added in version 3.5.

PyObject ***PyNumber_FloorDivide**(*PyObject* *o1, *PyObject* *o2)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return the floor of *o1* divided by *o2*, or `NULL` on failure. This is the equivalent of the Python expression `o1 // o2`.

PyObject ***PyNumber_TrueDivide**(*PyObject* *o1, *PyObject* *o2)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return a reasonable approximation for the mathematical value of *o1* divided by *o2*, or `NULL` on failure. The return value is „approximate" because binary floating-point numbers are approximate; it is not possible to represent all real numbers in base two. This function can return a floating-point value when passed two integers. This is the equivalent of the Python expression `o1 / o2`.

PyObject ***PyNumber_Remainder**(*PyObject* *o1, *PyObject* *o2)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Returns the remainder of dividing *o1* by *o2*, or `NULL` on failure. This is the equivalent of the Python expression `o1 % o2`.

PyObject ***PyNumber_Divmod**(*PyObject* *o1, *PyObject* *o2)

Wartość zwracana: nowa referencja. Część stabilnego ABI. See the built-in function `divmod()`. Returns `NULL` on failure. This is the equivalent of the Python expression `divmod(o1, o2)`.

PyObject ***PyNumber_Power**(*PyObject* *o1, *PyObject* *o2, *PyObject* *o3)

Wartość zwracana: nowa referencja. Część stabilnego ABI. See the built-in function `pow()`. Returns `NULL` on failure. This is the equivalent of the Python expression `pow(o1, o2, o3)`, where *o3* is optional. If *o3* is to be ignored, pass `Py_None` in its place (passing `NULL` for *o3* would cause an illegal memory access).

PyObject ***PyNumber_Negative**(*PyObject* *o)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Returns the negation of *o* on success, or `NULL` on failure. This is the equivalent of the Python expression `-o`.

PyObject ***PyNumber_Positive**(*PyObject* *o)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Returns *o* on success, or `NULL` on failure. This is the equivalent of the Python expression `+o`.

PyObject ***PyNumber_Absolute**(*PyObject* *o)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Returns the absolute value of *o*, or `NULL` on failure. This is the equivalent of the Python expression `abs(o)`.

PyObject ***PyNumber_Invert*** (*PyObject* *o)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Returns the bitwise negation of *o* on success, or `NULL` on failure. This is the equivalent of the Python expression `~o`.

PyObject ***PyNumber_Lshift*** (*PyObject* *o1, *PyObject* *o2)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Returns the result of left shifting *o1* by *o2* on success, or `NULL` on failure. This is the equivalent of the Python expression `o1 << o2`.

PyObject ***PyNumber_Rshift*** (*PyObject* *o1, *PyObject* *o2)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Returns the result of right shifting *o1* by *o2* on success, or `NULL` on failure. This is the equivalent of the Python expression `o1 >> o2`.

PyObject ***PyNumber_And*** (*PyObject* *o1, *PyObject* *o2)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Returns the „bitwise and" of *o1* and *o2* on success and `NULL` on failure. This is the equivalent of the Python expression `o1 & o2`.

PyObject ***PyNumber_Xor*** (*PyObject* *o1, *PyObject* *o2)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Returns the „bitwise exclusive or" of *o1* by *o2* on success, or `NULL` on failure. This is the equivalent of the Python expression `o1 ^ o2`.

PyObject ***PyNumber_Or*** (*PyObject* *o1, *PyObject* *o2)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Returns the „bitwise or" of *o1* and *o2* on success, or `NULL` on failure. This is the equivalent of the Python expression `o1 | o2`.

PyObject ***PyNumber_InPlaceAdd*** (*PyObject* *o1, *PyObject* *o2)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Returns the result of adding *o1* and *o2*, or `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 += o2`.

PyObject ***PyNumber_InPlaceSubtract*** (*PyObject* *o1, *PyObject* *o2)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Returns the result of subtracting *o2* from *o1*, or `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 -= o2`.

PyObject ***PyNumber_InPlaceMultiply*** (*PyObject* *o1, *PyObject* *o2)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Returns the result of multiplying *o1* and *o2*, or `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 *= o2`.

PyObject ***PyNumber_InPlaceMatrixMultiply*** (*PyObject* *o1, *PyObject* *o2)

Wartość zwracana: nowa referencja. Część stabilnego ABI *od wersji 3.7.* Returns the result of matrix multiplication on *o1* and *o2*, or `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 @= o2`.

Added in version 3.5.

PyObject ***PyNumber_InPlaceFloorDivide*** (*PyObject* *o1, *PyObject* *o2)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Returns the mathematical floor of dividing *o1* by *o2*, or `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 //= o2`.

PyObject ***PyNumber_InPlaceTrueDivide*** (*PyObject* *o1, *PyObject* *o2)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return a reasonable approximation for the mathematical value of *o1* divided by *o2*, or `NULL` on failure. The return value is „approximate" because binary floating-point numbers are approximate; it is not possible to represent all real numbers in base two. This function can return a floating-point value when passed two integers. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 /= o2`.

PyObject ***PyNumber_InPlaceRemainder*** (*PyObject* *o1, *PyObject* *o2)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Returns the remainder of dividing *o1* by *o2*, or `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 %= o2`.

PyObject ***PyNumber_InPlacePower*** (*PyObject* *o1, *PyObject* *o2, *PyObject* *o3)

Wartość zwracana: nowa referencja. Część stabilnego ABI. See the built-in function `pow()`. Returns `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 **= o2` when o3 is `Py_None`, or an in-place variant of `pow(o1, o2, o3)` otherwise. If *o3* is to be ignored, pass `Py_None` in its place (passing `NULL` for *o3* would cause an illegal memory access).

PyObject ***PyNumber_InPlaceLshift*** (*PyObject* *o1, *PyObject* *o2)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Returns the result of left shifting *o1* by *o2* on success, or `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 <<= o2`.

PyObject ***PyNumber_InPlaceRshift*** (*PyObject* *o1, *PyObject* *o2)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Returns the result of right shifting *o1* by *o2* on success, or `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 >>= o2`.

PyObject ***PyNumber_InPlaceAnd*** (*PyObject* *o1, *PyObject* *o2)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Returns the „bitwise and” of *o1* and *o2* on success and `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 &= o2`.

PyObject ***PyNumber_InPlaceXor*** (*PyObject* *o1, *PyObject* *o2)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Returns the „bitwise exclusive or” of *o1* by *o2* on success, or `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 ^= o2`.

PyObject ***PyNumber_InPlaceOr*** (*PyObject* *o1, *PyObject* *o2)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Returns the „bitwise or” of *o1* and *o2* on success, or `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 |= o2`.

PyObject ***PyNumber_Long*** (*PyObject* *o)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Returns the *o* converted to an integer object on success, or `NULL` on failure. This is the equivalent of the Python expression `int(o)`.

PyObject ***PyNumber_Float*** (*PyObject* *o)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Returns the *o* converted to a float object on success, or `NULL` on failure. This is the equivalent of the Python expression `float(o)`.

PyObject ***PyNumber_Index*** (*PyObject* *o)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Returns the *o* converted to a Python int on success or `NULL` with a `TypeError` exception raised on failure.

Zmienione w wersji 3.10: The result always has exact type `int`. Previously, the result could have been an instance of a subclass of `int`.

PyObject ***PyNumber_ToBase*** (*PyObject* *n, int base)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Returns the integer *n* converted to base *base* as a string. The *base* argument must be one of 2, 8, 10, or 16. For base 2, 8, or 16, the returned string is prefixed with a base marker of `'0b'`, `'0o'`, or `'0x'`, respectively. If *n* is not a Python int, it is converted with `PyNumber_Index()` first.

Py_ssize_t ***PyNumber_AsSsize_t*** (*PyObject* *o, *PyObject* *exc)

Część stabilnego ABI. Returns *o* converted to a `Py_ssize_t` value if *o* can be interpreted as an integer. If the call fails, an exception is raised and `-1` is returned.

If *o* can be converted to a Python int but the attempt to convert to a `Py_ssize_t` value would raise an `OverflowError`, then the *exc* argument is the type of exception that will be raised (usually `IndexError` or `OverflowError`). If *exc* is `NULL`, then the exception is cleared and the value is clipped to `PY_SSIZE_T_MIN` for a negative integer or `PY_SSIZE_T_MAX` for a positive integer.

int **PyIndex_Check**(*PyObject* *o)

Część stabilnego ABI *od wersji 3.8.* Returns `1` if *o* is an index integer (has the `nb_index` slot of the `tp_as_number` structure filled in), and `0` otherwise. This function always succeeds.

7.4 Sequence Protocol

int **PySequence_Check**(*PyObject* *o)

Część stabilnego ABI. Return `1` if the object provides the sequence protocol, and `0` otherwise. Note that it returns `1` for Python classes with a `__getitem__()` method, unless they are `dict` subclasses, since in general it is impossible to determine what type of keys the class supports. This function always succeeds.

Py_ssize_t **PySequence_Size**(*PyObject* *o)

Py_ssize_t **PySequence_Length**(*PyObject* *o)

Część stabilnego ABI. Returns the number of objects in sequence *o* on success, and `-1` on failure. This is equivalent to the Python expression `len(o)`.

PyObject ***PySequence_Concat**(*PyObject* *o1, *PyObject* *o2)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return the concatenation of *o1* and *o2* on success, and `NULL` on failure. This is the equivalent of the Python expression `o1 + o2`.

PyObject ***PySequence_Repeat**(*PyObject* *o, *Py_ssize_t* count)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return the result of repeating sequence object *o* *count* times, or `NULL` on failure. This is the equivalent of the Python expression `o * count`.

PyObject ***PySequence_InPlaceConcat**(*PyObject* *o1, *PyObject* *o2)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return the concatenation of *o1* and *o2* on success, and `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python expression `o1 += o2`.

PyObject ***PySequence_InPlaceRepeat**(*PyObject* *o, *Py_ssize_t* count)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return the result of repeating sequence object *o* *count* times, or `NULL` on failure. The operation is done *in-place* when *o* supports it. This is the equivalent of the Python expression `o *= count`.

PyObject ***PySequence_GetItem**(*PyObject* *o, *Py_ssize_t* i)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return the *i*th element of *o*, or `NULL` on failure. This is the equivalent of the Python expression `o[i]`.

PyObject ***PySequence_GetSlice**(*PyObject* *o, *Py_ssize_t* i1, *Py_ssize_t* i2)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return the slice of sequence object *o* between *i1* and *i2*, or `NULL` on failure. This is the equivalent of the Python expression `o[i1:i2]`.

int **PySequence_SetItem**(*PyObject* *o, *Py_ssize_t* i, *PyObject* *v)

Część stabilnego ABI. Assign object *v* to the *i*th element of *o*. Raise an exception and return `-1` on failure; return `0` on success. This is the equivalent of the Python statement `o[i] = v`. This function *does not* steal a reference to *v*.

If *v* is `NULL`, the element is deleted, but this feature is deprecated in favour of using `PySequence_DelItem()`.

int **PySequence_DelItem**(*PyObject* *o, *Py_ssize_t* i)

Część stabilnego ABI. Delete the *i*th element of object *o*. Returns `-1` on failure. This is the equivalent of the Python statement `del o[i]`.

int **PySequence_SetSlice**(*PyObject* *o, *Py_ssize_t* i1, *Py_ssize_t* i2, *PyObject* *v)

Część stabilnego ABI. Assign the sequence object *v* to the slice in sequence object *o* from *i1* to *i2*. This is the equivalent of the Python statement `o[i1:i2] = v`.

int **PySequence_DelSlice**(*PyObject* *o, *Py_ssize_t* i1, *Py_ssize_t* i2)

Część stabilnego ABI. Delete the slice in sequence object *o* from *i1* to *i2*. Returns `-1` on failure. This is the equivalent of the Python statement `del o[i1:i2]`.

Py_ssize_t **PySequence_Count**(*PyObject* *o, *PyObject* *value)

Część stabilnego ABI. Return the number of occurrences of *value* in *o*, that is, return the number of keys for which `o[key] == value`. On failure, return `-1`. This is equivalent to the Python expression `o.count(value)`.

int **PySequence_Contains**(*PyObject* *o, *PyObject* *value)

Część stabilnego ABI. Determine if *o* contains *value*. If an item in *o* is equal to *value*, return `1`, otherwise return `0`. On error, return `-1`. This is equivalent to the Python expression `value in o`.

Py_ssize_t **PySequence_Index**(*PyObject* *o, *PyObject* *value)

Część stabilnego ABI. Return the first index *i* for which `o[i] == value`. On error, return `-1`. This is equivalent to the Python expression `o.index(value)`.

PyObject ***PySequence_List**(*PyObject* *o)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return a list object with the same contents as the sequence or iterable *o*, or `NULL` on failure. The returned list is guaranteed to be new. This is equivalent to the Python expression `list(o)`.

PyObject ***PySequence_Tuple**(*PyObject* *o)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return a tuple object with the same contents as the sequence or iterable *o*, or `NULL` on failure. If *o* is a tuple, a new reference will be returned, otherwise a tuple will be constructed with the appropriate contents. This is equivalent to the Python expression `tuple(o)`.

PyObject ***PySequence_Fast**(*PyObject* *o, const char *m)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return the sequence or iterable *o* as an object usable by the other `PySequence_Fast*` family of functions. If the object is not a sequence or iterable, raises `TypeError` with *m* as the message text. Returns `NULL` on failure.

The `PySequence_Fast*` functions are thus named because they assume *o* is a `PyTupleObject` or a `PyListObject` and access the data fields of *o* directly.

As a CPython implementation detail, if *o* is already a sequence or list, it will be returned.

Py_ssize_t **PySequence_Fast_GET_SIZE**(*PyObject* *o)

Returns the length of *o*, assuming that *o* was returned by `PySequence_Fast()` and that *o* is not `NULL`. The size can also be retrieved by calling `PySequence_Size()` on *o*, but `PySequence_Fast_GET_SIZE()` is faster because it can assume *o* is a list or tuple.

PyObject ***PySequence_Fast_GET_ITEM**(*PyObject* *o, *Py_ssize_t* i)

Wartość zwracana: pożyczona referencja. Return the *i*th element of *o*, assuming that *o* was returned by `PySequence_Fast()`, *o* is not `NULL`, and that *i* is within bounds.

PyObject ****PySequence_Fast_ITEMS**(*PyObject* *o)

Return the underlying array of PyObject pointers. Assumes that *o* was returned by `PySequence_Fast()` and *o* is not `NULL`.

Note, if a list gets resized, the reallocation may relocate the items array. So, only use the underlying array pointer in contexts where the sequence cannot change.

PyObject ***PySequence_ITEM**(*PyObject* *o, *Py_ssize_t* i)

Wartość zwracana: nowa referencja. Return the *i*th element of *o* or `NULL` on failure. Faster form of `PySequence_GetItem()` but without checking that `PySequence_Check()` on *o* is true and without adjustment for negative indices.

7.5 Mapping Protocol

See also `PyObject_GetItem()`, `PyObject_SetItem()` and `PyObject_DelItem()`.

int **PyMapping_Check**(*PyObject* *o)

Część stabilnego ABI. Return `1` if the object provides the mapping protocol or supports slicing, and `0` otherwise. Note that it returns `1` for Python classes with a `__getitem__()` method, since in general it is impossible to determine what type of keys the class supports. This function always succeeds.

Py_ssize_t **PyMapping_Size**(*PyObject* *o)

Py_ssize_t **PyMapping_Length**(*PyObject* *o)

Część stabilnego ABI. Returns the number of keys in object *o* on success, and `-1` on failure. This is equivalent to the Python expression `len(o)`.

PyObject ***PyMapping_GetItemString**(*PyObject* *o, const char *key)

Wartość zwracana: nowa referencja. Część stabilnego ABI. This is the same as `PyObject_GetItem()`, but *key* is specified as a `const char*` UTF-8 encoded bytes string, rather than a `PyObject*`.

int **PyMapping_GetOptionalItem**(*PyObject* *obj, *PyObject* *key, *PyObject* **result)

Część stabilnego ABI *od wersji 3.13.* Variant of `PyObject_GetItem()` which doesn't raise `KeyError` if the key is not found.

If the key is found, return `1` and set **result* to a new *strong reference* to the corresponding value. If the key is not found, return `0` and set **result* to `NULL`; the `KeyError` is silenced. If an error other than `KeyError` is raised, return `-1` and set **result* to `NULL`.

Added in version 3.13.

int **PyMapping_GetOptionalItemString**(*PyObject* *obj, const char *key, *PyObject* **result)

Część stabilnego ABI *od wersji 3.13.* This is the same as `PyMapping_GetOptionalItem()`, but *key* is specified as a `const char*` UTF-8 encoded bytes string, rather than a `PyObject*`.

Added in version 3.13.

int **PyMapping_SetItemString**(*PyObject* *o, const char *key, *PyObject* *v)

Część stabilnego ABI. This is the same as `PyObject_SetItem()`, but *key* is specified as a `const char*` UTF-8 encoded bytes string, rather than a `PyObject*`.

int **PyMapping_DelItem**(*PyObject* *o, *PyObject* *key)

This is an alias of `PyObject_DelItem()`.

int **PyMapping_DelItemString**(*PyObject* *o, const char *key)

This is the same as `PyObject_DelItem()`, but *key* is specified as a `const char*` UTF-8 encoded bytes string, rather than a `PyObject*`.

int **PyMapping_HasKeyWithError**(*PyObject* *o, *PyObject* *key)

Część stabilnego ABI *od wersji 3.13.* Return `1` if the mapping object has the key *key* and `0` otherwise. This is equivalent to the Python expression `key in o`. On failure, return `-1`.

Added in version 3.13.

int **PyMapping_HasKeyStringWithError**(*PyObject* *o, const char *key)

Część stabilnego ABI *od wersji 3.13.* This is the same as `PyMapping_HasKeyWithError()`, but *key* is specified as a `const char*` UTF-8 encoded bytes string, rather than a `PyObject*`.

Added in version 3.13.

int **PyMapping_HasKey**(*PyObject* *o, *PyObject* *key)

Część stabilnego ABI. Return `1` if the mapping object has the key *key* and `0` otherwise. This is equivalent to the Python expression `key in o`. This function always succeeds.

> **Informacja**
>
> Exceptions which occur when this calls `__getitem__()` method are silently ignored. For proper error handling, use `PyMapping_HasKeyWithError()`, `PyMapping_GetOptionalItem()` or `PyObject_GetItem()` instead.

int **PyMapping_HasKeyString**(*PyObject* *o, const char *key)

Część stabilnego ABI. This is the same as `PyMapping_HasKey()`, but *key* is specified as a `const char*` UTF-8 encoded bytes string, rather than a `PyObject*`.

> **Informacja**
>
> Exceptions that occur when this calls `__getitem__()` method or while creating the temporary `str` object are silently ignored. For proper error handling, use `PyMapping_HasKeyStringWithError()`, `PyMapping_GetOptionalItemString()` or `PyMapping_GetItemString()` instead.

PyObject ***PyMapping_Keys**(*PyObject* *o)

Wartość zwracana: nowa referencja. Część stabilnego ABI. On success, return a list of the keys in object *o*. On failure, return `NULL`.

Zmienione w wersji 3.7: Previously, the function returned a list or a tuple.

PyObject ***PyMapping_Values**(*PyObject* *o)

Wartość zwracana: nowa referencja. Część stabilnego ABI. On success, return a list of the values in object *o*. On failure, return `NULL`.

Zmienione w wersji 3.7: Previously, the function returned a list or a tuple.

PyObject ***PyMapping_Items**(*PyObject* *o)

Wartość zwracana: nowa referencja. Część stabilnego ABI. On success, return a list of the items in object *o*, where each item is a tuple containing a key-value pair. On failure, return `NULL`.

Zmienione w wersji 3.7: Previously, the function returned a list or a tuple.

7.6 Iterator Protocol

There are two functions specifically for working with iterators.

int **PyIter_Check**(*PyObject* *o)

Część stabilnego ABI *od wersji 3.8.* Return non-zero if the object *o* can be safely passed to `PyIter_Next()`, and `0` otherwise. This function always succeeds.

int **PyAIter_Check**(*PyObject* *o)

Część stabilnego ABI *od wersji 3.10.* Return non-zero if the object *o* provides the `AsyncIterator` protocol, and `0` otherwise. This function always succeeds.

Added in version 3.10.

PyObject ***PyIter_Next**(*PyObject* *o)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return the next value from the iterator *o*. The object must be an iterator according to `PyIter_Check()` (it is up to the caller to check this). If there are no remaining values, returns `NULL` with no exception set. If an error occurs while retrieving the item, returns `NULL` and passes along the exception.

To write a loop which iterates over an iterator, the C code should look something like this:

```
PyObject *iterator = PyObject_GetIter(obj);
PyObject *item;

if (iterator == NULL) {
    /* propagate error */
}

while ((item = PyIter_Next(iterator))) {
    /* do something with item */
    ...
    /* release reference when done */
    Py_DECREF(item);
}
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
Py_DECREF(iterator);

if (PyErr_Occurred()) {
    /* propagate error */
}
else {
    /* continue doing useful work */
}
```

type **PySendResult**

The enum value used to represent different results of `PyIter_Send()`.

Added in version 3.10.

PySendResult **PyIter_Send**(*PyObject* *iter, *PyObject* *arg, *PyObject* **presult)

Część stabilnego ABI *od wersji 3.10.* Sends the *arg* value into the iterator *iter*. Returns:

- `PYGEN_RETURN` if iterator returns. Return value is returned via *presult*.
- `PYGEN_NEXT` if iterator yields. Yielded value is returned via *presult*.
- `PYGEN_ERROR` if iterator has raised and exception. *presult* is set to `NULL`.

Added in version 3.10.

7.7 Buffer Protocol

Certain objects available in Python wrap access to an underlying memory array or *buffer*. Such objects include the built-in `bytes` and `bytearray`, and some extension types like `array.array`. Third-party libraries may define their own types for special purposes, such as image processing or numeric analysis.

While each of these types have their own semantics, they share the common characteristic of being backed by a possibly large memory buffer. It is then desirable, in some situations, to access that buffer directly and without intermediate copying.

Python provides such a facility at the C level in the form of the *buffer protocol*. This protocol has two sides:

- on the producer side, a type can export a „buffer interface" which allows objects of that type to expose information about their underlying buffer. This interface is described in the section *Buffer Object Structures*;
- on the consumer side, several means are available to obtain a pointer to the raw underlying data of an object (for example a method parameter).

Simple objects such as `bytes` and `bytearray` expose their underlying buffer in byte-oriented form. Other forms are possible; for example, the elements exposed by an `array.array` can be multi-byte values.

An example consumer of the buffer interface is the `write()` method of file objects: any object that can export a series of bytes through the buffer interface can be written to a file. While `write()` only needs read-only access to the internal contents of the object passed to it, other methods such as `readinto()` need write access to the contents of their argument. The buffer interface allows objects to selectively allow or reject exporting of read-write and read-only buffers.

There are two ways for a consumer of the buffer interface to acquire a buffer over a target object:

- call `PyObject_GetBuffer()` with the right parameters;
- call `PyArg_ParseTuple()` (or one of its siblings) with one of the `y*`, `w*` or `s*` *format codes*.

In both cases, `PyBuffer_Release()` must be called when the buffer isn't needed anymore. Failure to do so could lead to various issues such as resource leaks.

7.7.1 Buffer structure

Buffer structures (or simply „buffers") are useful as a way to expose the binary data from another object to the Python programmer. They can also be used as a zero-copy slicing mechanism. Using their ability to reference a block of memory, it is possible to expose any data to the Python programmer quite easily. The memory could be a large, constant array in a C extension, it could be a raw block of memory for manipulation before passing to an operating system library, or it could be used to pass around structured data in its native, in-memory format.

Contrary to most data types exposed by the Python interpreter, buffers are not `PyObject` pointers but rather simple C structures. This allows them to be created and copied very simply. When a generic wrapper around a buffer is needed, a *memoryview* object can be created.

For short instructions how to write an exporting object, see *Buffer Object Structures*. For obtaining a buffer, see `PyObject_GetBuffer()`.

type **Py_buffer**

Część stabilnego ABI *(w tym wszystkie składniki) od wersji 3.11.*

void ***buf**

A pointer to the start of the logical structure described by the buffer fields. This can be any location within the underlying physical memory block of the exporter. For example, with negative `strides` the value may point to the end of the memory block.

For *contiguous* arrays, the value points to the beginning of the memory block.

PyObject ***obj**

A new reference to the exporting object. The reference is owned by the consumer and automatically released (i.e. reference count decremented) and set to `NULL` by `PyBuffer_Release()`. The field is the equivalent of the return value of any standard C-API function.

As a special case, for *temporary* buffers that are wrapped by `PyMemoryView_FromBuffer()` or `PyBuffer_FillInfo()` this field is `NULL`. In general, exporting objects MUST NOT use this scheme.

Py_ssize_t **len**

`product(shape) * itemsize`. For contiguous arrays, this is the length of the underlying memory block. For non-contiguous arrays, it is the length that the logical structure would have if it were copied to a contiguous representation.

Accessing `((char *)buf)[0] up to ((char *)buf)[len-1]` is only valid if the buffer has been obtained by a request that guarantees contiguity. In most cases such a request will be `PyBUF_SIMPLE` or `PyBUF_WRITABLE`.

int **readonly**

An indicator of whether the buffer is read-only. This field is controlled by the `PyBUF_WRITABLE` flag.

Py_ssize_t **itemsize**

Item size in bytes of a single element. Same as the value of `struct.calcsize()` called on non-`NULL` `format` values.

Important exception: If a consumer requests a buffer without the `PyBUF_FORMAT` flag, `format` will be set to `NULL`, but `itemsize` still has the value for the original format.

If `shape` is present, the equality `product(shape) * itemsize == len` still holds and the consumer can use `itemsize` to navigate the buffer.

If `shape` is `NULL` as a result of a `PyBUF_SIMPLE` or a `PyBUF_WRITABLE` request, the consumer must disregard `itemsize` and assume `itemsize == 1`.

char ***format**

A *NULL* terminated string in `struct` module style syntax describing the contents of a single item. If this is `NULL`, `"B"` (unsigned bytes) is assumed.

This field is controlled by the `PyBUF_FORMAT` flag.

int **ndim**

The number of dimensions the memory represents as an n-dimensional array. If it is `0`, `buf` points to a single item representing a scalar. In this case, `shape`, `strides` and `suboffsets` MUST be `NULL`. The maximum number of dimensions is given by `PyBUF_MAX_NDIM`.

Py_ssize_t ***shape**

An array of `Py_ssize_t` of length `ndim` indicating the shape of the memory as an n-dimensional array. Note that `shape[0] * ... * shape[ndim-1] * itemsize` MUST be equal to `len`.

Shape values are restricted to `shape[n] >= 0`. The case `shape[n] == 0` requires special attention. See *complex arrays* for further information.

The shape array is read-only for the consumer.

Py_ssize_t ***strides**

An array of `Py_ssize_t` of length `ndim` giving the number of bytes to skip to get to a new element in each dimension.

Stride values can be any integer. For regular arrays, strides are usually positive, but a consumer MUST be able to handle the case `strides[n] <= 0`. See *complex arrays* for further information.

The strides array is read-only for the consumer.

Py_ssize_t ***suboffsets**

An array of `Py_ssize_t` of length `ndim`. If `suboffsets[n] >= 0`, the values stored along the nth dimension are pointers and the suboffset value dictates how many bytes to add to each pointer after de--referencing. A suboffset value that is negative indicates that no de-referencing should occur (striding in a contiguous memory block).

If all suboffsets are negative (i.e. no de-referencing is needed), then this field must be `NULL` (the default value).

This type of array representation is used by the Python Imaging Library (PIL). See *complex arrays* for further information how to access elements of such an array.

The suboffsets array is read-only for the consumer.

void ***internal**

This is for use internally by the exporting object. For example, this might be re-cast as an integer by the exporter and used to store flags about whether or not the shape, strides, and suboffsets arrays must be freed when the buffer is released. The consumer MUST NOT alter this value.

Constants:

PyBUF_MAX_NDIM

The maximum number of dimensions the memory represents. Exporters MUST respect this limit, consumers of multi-dimensional buffers SHOULD be able to handle up to `PyBUF_MAX_NDIM` dimensions. Currently set to 64.

7.7.2 Buffer request types

Buffers are usually obtained by sending a buffer request to an exporting object via `PyObject_GetBuffer()`. Since the complexity of the logical structure of the memory can vary drastically, the consumer uses the *flags* argument to specify the exact buffer type it can handle.

All `Py_buffer` fields are unambiguously defined by the request type.

request-independent fields

The following fields are not influenced by *flags* and must always be filled in with the correct values: `obj`, `buf`, `len`, `itemsize`, `ndim`.

readonly, format

PyBUF_WRITABLE

Controls the *readonly* field. If set, the exporter MUST provide a writable buffer or else report failure. Otherwise, the exporter MAY provide either a read-only or writable buffer, but the choice MUST be consistent for all consumers. For example, *PyBUF_SIMPLE* | *PyBUF_WRITABLE* can be used to request a simple writable buffer.

PyBUF_FORMAT

Controls the *format* field. If set, this field MUST be filled in correctly. Otherwise, this field MUST be NULL.

PyBUF_WRITABLE can be |»d to any of the flags in the next section. Since *PyBUF_SIMPLE* is defined as 0, *PyBUF_WRITABLE* can be used as a stand-alone flag to request a simple writable buffer.

PyBUF_FORMAT must be |»d to any of the flags except *PyBUF_SIMPLE*, because the latter already implies format B (unsigned bytes). PyBUF_FORMAT cannot be used on its own.

shape, strides, suboffsets

The flags that control the logical structure of the memory are listed in decreasing order of complexity. Note that each flag contains all bits of the flags below it.

Request	shape	strides	suboffsets
PyBUF_INDIRECT	tak	tak	if needed
PyBUF_STRIDES	tak	tak	NULL
PyBUF_ND	tak	NULL	NULL
PyBUF_SIMPLE	NULL	NULL	NULL

contiguity requests

C or Fortran *contiguity* can be explicitly requested, with and without stride information. Without stride information, the buffer must be C-contiguous.

Request	shape	strides	suboffsets	contig
PyBUF_C_CONTIGUOUS	tak	tak	NULL	C
PyBUF_F_CONTIGUOUS	tak	tak	NULL	F
PyBUF_ANY_CONTIGUOUS	tak	tak	NULL	C or F
PyBUF_ND	tak	NULL	NULL	C

compound requests

All possible requests are fully defined by some combination of the flags in the previous section. For convenience, the buffer protocol provides frequently used combinations as single flags.

In the following table *U* stands for undefined contiguity. The consumer would have to call `PyBuffer_IsContiguous()` to determine contiguity.

Request	shape	strides	suboffsets	contig	readonly	format
PyBUF_FULL	tak	tak	if needed	U	0	tak
PyBUF_FULL_RO	tak	tak	if needed	U	1 or 0	tak
PyBUF_RECORDS	tak	tak	NULL	U	0	tak
PyBUF_RECORDS_RO	tak	tak	NULL	U	1 or 0	tak
PyBUF_STRIDED	tak	tak	NULL	U	0	NULL
PyBUF_STRIDED_RO	tak	tak	NULL	U	1 or 0	NULL
PyBUF_CONTIG	tak	NULL	NULL	C	0	NULL
PyBUF_CONTIG_RO	tak	NULL	NULL	C	1 or 0	NULL

7.7.3 Complex arrays

NumPy-style: shape and strides

The logical structure of NumPy-style arrays is defined by `itemsize`, `ndim`, `shape` and `strides`.

If `ndim == 0`, the memory location pointed to by `buf` is interpreted as a scalar of size `itemsize`. In that case, both `shape` and `strides` are `NULL`.

If `strides` is `NULL`, the array is interpreted as a standard n-dimensional C-array. Otherwise, the consumer must access an n-dimensional array as follows:

```
ptr = (char *)buf + indices[0] * strides[0] + ... + indices[n-1] * strides[n-1];
item = *((typeof(item) *)ptr);
```

As noted above, `buf` can point to any location within the actual memory block. An exporter can check the validity of a buffer with this function:

```
def verify_structure(memlen, itemsize, ndim, shape, strides, offset):
    """Verify that the parameters represent a valid array within
       the bounds of the allocated memory:
           char *mem: start of the physical memory block
           memlen: length of the physical memory block
           offset: (char *)buf - mem
    """
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
    if offset % itemsize:
        return False
    if offset < 0 or offset+itemsize > memlen:
        return False
    if any(v % itemsize for v in strides):
        return False

    if ndim <= 0:
        return ndim == 0 and not shape and not strides
    if 0 in shape:
        return True

    imin = sum(strides[j]*(shape[j]-1) for j in range(ndim)
               if strides[j] <= 0)
    imax = sum(strides[j]*(shape[j]-1) for j in range(ndim)
               if strides[j] > 0)

    return 0 <= offset+imin and offset+imax+itemsize <= memlen
```

PIL-style: shape, strides and suboffsets

In addition to the regular items, PIL-style arrays can contain pointers that must be followed in order to get to the next element in a dimension. For example, the regular three-dimensional C-array `char v[2][2][3]` can also be viewed as an array of 2 pointers to 2 two-dimensional arrays: `char (*v[2])[2][3]`. In suboffsets representation, those two pointers can be embedded at the start of *buf*, pointing to two `char x[2][3]` arrays that can be located anywhere in memory.

Here is a function that returns a pointer to the element in an N-D array pointed to by an N-dimensional index when there are both non-`NULL` strides and suboffsets:

```
void *get_item_pointer(int ndim, void *buf, Py_ssize_t *strides,
                       Py_ssize_t *suboffsets, Py_ssize_t *indices) {
    char *pointer = (char*)buf;
    int i;
    for (i = 0; i < ndim; i++) {
        pointer += strides[i] * indices[i];
        if (suboffsets[i] >=0 ) {
            pointer = *((char**)pointer) + suboffsets[i];
        }
    }
    return (void*)pointer;
}
```

7.7.4 Buffer-related functions

int **PyObject_CheckBuffer**(*PyObject* *obj)

Część stabilnego ABI *od wersji 3.11*. Return `1` if *obj* supports the buffer interface otherwise `0`. When `1` is returned, it doesn't guarantee that `PyObject_GetBuffer()` will succeed. This function always succeeds.

int **PyObject_GetBuffer**(*PyObject* *exporter, *Py_buffer* *view, int flags)

Część stabilnego ABI *od wersji 3.11*. Send a request to *exporter* to fill in *view* as specified by *flags*. If the exporter cannot provide a buffer of the exact type, it MUST raise `BufferError`, set `view->obj` to `NULL` and return `-1`.

On success, fill in *view*, set `view->obj` to a new reference to *exporter* and return 0. In the case of chained buffer providers that redirect requests to a single object, `view->obj` MAY refer to this object instead of *exporter* (See *Buffer Object Structures*).

Successful calls to `PyObject_GetBuffer()` must be paired with calls to `PyBuffer_Release()`, similar to `malloc()` and `free()`. Thus, after the consumer is done with the buffer, `PyBuffer_Release()` must be called exactly once.

void **PyBuffer_Release**(*Py_buffer* *view)

Część stabilnego ABI *od wersji 3.11.* Release the buffer *view* and release the *strong reference* (i.e. decrement the reference count) to the view's supporting object, `view->obj`. This function MUST be called when the buffer is no longer being used, otherwise reference leaks may occur.

It is an error to call this function on a buffer that was not obtained via `PyObject_GetBuffer()`.

Py_ssize_t **PyBuffer_SizeFromFormat**(const char *format)

Część stabilnego ABI *od wersji 3.11.* Return the implied `itemsize` from `format`. On error, raise an exception and return -1.

Added in version 3.9.

int **PyBuffer_IsContiguous**(const *Py_buffer* *view, char order)

Część stabilnego ABI *od wersji 3.11.* Return `1` if the memory defined by the *view* is C-style (*order* is `'C'`) or Fortran-style (*order* is `'F'`) *contiguous* or either one (*order* is `'A'`). Return `0` otherwise. This function always succeeds.

void ***PyBuffer_GetPointer**(const *Py_buffer* *view, const *Py_ssize_t* *indices)

Część stabilnego ABI *od wersji 3.11.* Get the memory area pointed to by the *indices* inside the given *view*. *indices* must point to an array of `view->ndim` indices.

int **PyBuffer_FromContiguous**(const *Py_buffer* *view, const void *buf, *Py_ssize_t* len, char fort)

Część stabilnego ABI *od wersji 3.11.* Copy contiguous *len* bytes from *buf* to *view*. *fort* can be `'C'` or `'F'` (for C-style or Fortran-style ordering). `0` is returned on success, `-1` on error.

int **PyBuffer_ToContiguous**(void *buf, const *Py_buffer* *src, *Py_ssize_t* len, char order)

Część stabilnego ABI *od wersji 3.11.* Copy *len* bytes from *src* to its contiguous representation in *buf*. *order* can be `'C'` or `'F'` or `'A'` (for C-style or Fortran-style ordering or either one). `0` is returned on success, `-1` on error.

This function fails if *len* != *src->len*.

int **PyObject_CopyData**(*PyObject* *dest, *PyObject* *src)

Część stabilnego ABI *od wersji 3.11.* Copy data from *src* to *dest* buffer. Can convert between C-style and or Fortran-style buffers.

`0` is returned on success, `-1` on error.

void **PyBuffer_FillContiguousStrides**(int ndims, *Py_ssize_t* *shape, *Py_ssize_t* *strides, int itemsize, char order)

Część stabilnego ABI *od wersji 3.11.* Fill the *strides* array with byte-strides of a *contiguous* (C-style if *order* is `'C'` or Fortran-style if *order* is `'F'`) array of the given shape with the given number of bytes per element.

int **PyBuffer_FillInfo**(*Py_buffer* *view, *PyObject* *exporter, void *buf, *Py_ssize_t* len, int readonly, int flags)

Część stabilnego ABI *od wersji 3.11.* Handle buffer requests for an exporter that wants to expose *buf* of size *len* with writability set according to *readonly*. *buf* is interpreted as a sequence of unsigned bytes.

The *flags* argument indicates the request type. This function always fills in *view* as specified by flags, unless *buf* has been designated as read-only and `PyBUF_WRITABLE` is set in *flags*.

On success, set `view->obj` to a new reference to *exporter* and return 0. Otherwise, raise `BufferError`, set `view->obj` to `NULL` and return `-1`;

If this function is used as part of a *getbufferproc*, *exporter* MUST be set to the exporting object and *flags* must be passed unmodified. Otherwise, *exporter* MUST be `NULL`.

ROZDZIAŁ 8

Concrete Objects Layer

The functions in this chapter are specific to certain Python object types. Passing them an object of the wrong type is not a good idea; if you receive an object from a Python program and you are not sure that it has the right type, you must perform a type check first; for example, to check that an object is a dictionary, use `PyDict_Check()`. The chapter is structured like the „family tree" of Python object types.

> **Ostrzeżenie**
>
> While the functions described in this chapter carefully check the type of the objects which are passed in, many of them do not check for `NULL` being passed instead of a valid object. Allowing `NULL` to be passed in can cause memory access violations and immediate termination of the interpreter.

8.1 Fundamental Objects

This section describes Python type objects and the singleton object `None`.

8.1.1 Type Objects

type **`PyTypeObject`**

Część ograniczonego API *(jako nieprzezroczysta struktura)*. The C structure of the objects used to describe built-in types.

PyTypeObject **`PyType_Type`**

Część stabilnego ABI. This is the type object for type objects; it is the same object as `type` in the Python layer.

int **`PyType_Check`**(*PyObject* *o)

Return non-zero if the object *o* is a type object, including instances of types derived from the standard type object. Return 0 in all other cases. This function always succeeds.

int **`PyType_CheckExact`**(*PyObject* *o)

Return non-zero if the object *o* is a type object, but not a subtype of the standard type object. Return 0 in all other cases. This function always succeeds.

unsigned int **`PyType_ClearCache`**()

Część stabilnego ABI. Clear the internal lookup cache. Return the current version tag.

unsigned long **PyType_GetFlags**(*PyTypeObject* *type)

Część stabilnego ABI. Return the `tp_flags` member of *type*. This function is primarily meant for use with `Py_LIMITED_API`; the individual flag bits are guaranteed to be stable across Python releases, but access to `tp_flags` itself is not part of the *limited API*.

Added in version 3.2.

Zmienione w wersji 3.4: The return type is now `unsigned long` rather than `long`.

PyObject ***PyType_GetDict**(*PyTypeObject* *type)

Return the type object's internal namespace, which is otherwise only exposed via a read-only proxy (`cls.__dict__`). This is a replacement for accessing `tp_dict` directly. The returned dictionary must be treated as read-only.

This function is meant for specific embedding and language-binding cases, where direct access to the dict is necessary and indirect access (e.g. via the proxy or `PyObject_GetAttr()`) isn't adequate.

Extension modules should continue to use `tp_dict`, directly or indirectly, when setting up their own types.

Added in version 3.12.

void **PyType_Modified**(*PyTypeObject* *type)

Część stabilnego ABI. Invalidate the internal lookup cache for the type and all of its subtypes. This function must be called after any manual modification of the attributes or base classes of the type.

int **PyType_AddWatcher**(*PyType_WatchCallback* callback)

Register *callback* as a type watcher. Return a non-negative integer ID which must be passed to future calls to `PyType_Watch()`. In case of error (e.g. no more watcher IDs available), return `-1` and set an exception.

Added in version 3.12.

int **PyType_ClearWatcher**(int watcher_id)

Clear watcher identified by *watcher_id* (previously returned from `PyType_AddWatcher()`). Return `0` on success, `-1` on error (e.g. if *watcher_id* was never registered.)

An extension should never call `PyType_ClearWatcher` with a *watcher_id* that was not returned to it by a previous call to `PyType_AddWatcher()`.

Added in version 3.12.

int **PyType_Watch**(int watcher_id, *PyObject* *type)

Mark *type* as watched. The callback granted *watcher_id* by `PyType_AddWatcher()` will be called whenever `PyType_Modified()` reports a change to *type*. (The callback may be called only once for a series of consecutive modifications to *type*, if `_PyType_Lookup()` is not called on *type* between the modifications; this is an implementation detail and subject to change.)

An extension should never call `PyType_Watch` with a *watcher_id* that was not returned to it by a previous call to `PyType_AddWatcher()`.

Added in version 3.12.

typedef int (***PyType_WatchCallback**)(*PyObject* *type)

Type of a type-watcher callback function.

The callback must not modify *type* or cause `PyType_Modified()` to be called on *type* or any type in its MRO; violating this rule could cause infinite recursion.

Added in version 3.12.

int **PyType_HasFeature**(*PyTypeObject* *o, int feature)

Return non-zero if the type object *o* sets the feature *feature*. Type features are denoted by single bit flags.

int **PyType_IS_GC**(*PyTypeObject* *o)

Return true if the type object includes support for the cycle detector; this tests the type flag `Py_TPFLAGS_HAVE_GC`.

int **PyType_IsSubtype**(*PyTypeObject* *a, *PyTypeObject* *b)

Część stabilnego ABI. Return true if *a* is a subtype of *b*.

This function only checks for actual subtypes, which means that `__subclasscheck__()` is not called on *b*. Call `PyObject_IsSubclass()` to do the same check that `issubclass()` would do.

PyObject ***PyType_GenericAlloc**(*PyTypeObject* *type, *Py_ssize_t* nitems)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Generic handler for the `tp_alloc` slot of a type object. Use Python's default memory allocation mechanism to allocate a new instance and initialize all its contents to `NULL`.

PyObject ***PyType_GenericNew**(*PyTypeObject* *type, *PyObject* *args, *PyObject* *kwds)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Generic handler for the `tp_new` slot of a type object. Create a new instance using the type's `tp_alloc` slot.

int **PyType_Ready**(*PyTypeObject* *type)

Część stabilnego ABI. Finalize a type object. This should be called on all type objects to finish their initialization. This function is responsible for adding inherited slots from a type's base class. Return `0` on success, or return `-1` and sets an exception on error.

> **Informacja**
>
> If some of the base classes implements the GC protocol and the provided type does not include the `Py_TPFLAGS_HAVE_GC` in its flags, then the GC protocol will be automatically implemented from its parents. On the contrary, if the type being created does include `Py_TPFLAGS_HAVE_GC` in its flags then it **must** implement the GC protocol itself by at least implementing the `tp_traverse` handle.

PyObject ***PyType_GetName**(*PyTypeObject* *type)

Wartość zwracana: nowa referencja. Część stabilnego ABI *od wersji 3.11.* Return the type's name. Equivalent to getting the type's `__name__` attribute.

Added in version 3.11.

PyObject ***PyType_GetQualName**(*PyTypeObject* *type)

Wartość zwracana: nowa referencja. Część stabilnego ABI *od wersji 3.11.* Return the type's qualified name. Equivalent to getting the type's `__qualname__` attribute.

Added in version 3.11.

PyObject ***PyType_GetFullyQualifiedName**(*PyTypeObject* *type)

Część stabilnego ABI *od wersji 3.13.* Return the type's fully qualified name. Equivalent to `f"{type.__module__}.{type.__qualname__}"`, or `type.__qualname__` if `type.__module__` is not a string or is equal to `"builtins"`.

Added in version 3.13.

PyObject ***PyType_GetModuleName**(*PyTypeObject* *type)

Część stabilnego ABI *od wersji 3.13.* Return the type's module name. Equivalent to getting the `type.__module__` attribute.

Added in version 3.13.

void ***PyType_GetSlot**(*PyTypeObject* *type, int slot)

Część stabilnego ABI *od wersji 3.4.* Return the function pointer stored in the given slot. If the result is `NULL`, this indicates that either the slot is `NULL`, or that the function was called with invalid parameters. Callers will typically cast the result pointer into the appropriate function type.

See `PyType_Slot.slot` for possible values of the *slot* argument.

Added in version 3.4.

Zmienione w wersji 3.10: `PyType_GetSlot()` can now accept all types. Previously, it was limited to *heap types*.

PyObject ***PyType_GetModule**(*PyTypeObject* *type)

Część stabilnego ABI *od wersji 3.10.* Return the module object associated with the given type when the type was created using `PyType_FromModuleAndSpec()`.

If no module is associated with the given type, sets `TypeError` and returns `NULL`.

This function is usually used to get the module in which a method is defined. Note that in such a method, `PyType_GetModule(Py_TYPE(self))` may not return the intended result. `Py_TYPE(self)` may be a *subclass* of the intended class, and subclasses are not necessarily defined in the same module as their superclass. See `PyCMethod` to get the class that defines the method. See `PyType_GetModuleByDef()` for cases when `PyCMethod` cannot be used.

Added in version 3.9.

void ***PyType_GetModuleState**(*PyTypeObject* *type)

Część stabilnego ABI *od wersji 3.10.* Return the state of the module object associated with the given type. This is a shortcut for calling `PyModule_GetState()` on the result of `PyType_GetModule()`.

If no module is associated with the given type, sets `TypeError` and returns `NULL`.

If the *type* has an associated module but its state is `NULL`, returns `NULL` without setting an exception.

Added in version 3.9.

PyObject ***PyType_GetModuleByDef**(*PyTypeObject* *type, struct *PyModuleDef* *def)

Część stabilnego ABI *od wersji 3.13.* Find the first superclass whose module was created from the given `PyModuleDef` *def*, and return that module.

If no module is found, raises a `TypeError` and returns `NULL`.

This function is intended to be used together with `PyModule_GetState()` to get module state from slot methods (such as `tp_init` or `nb_add`) and other places where a method's defining class cannot be passed using the `PyCMethod` calling convention.

Added in version 3.11.

int **PyUnstable_Type_AssignVersionTag**(*PyTypeObject* *type)

> To jest *niestabilne API*. Może ulec zmianie bez ostrzeżenia w wydaniach minor.

Attempt to assign a version tag to the given type.

Returns 1 if the type already had a valid version tag or a new one was assigned, or 0 if a new tag could not be assigned.

Added in version 3.12.

Creating Heap-Allocated Types

The following functions and structs are used to create *heap types*.

PyObject ***PyType_FromMetaclass**(*PyTypeObject* *metaclass, *PyObject* *module, *PyType_Spec* *spec, *PyObject* *bases)

Część stabilnego ABI *od wersji 3.12.* Create and return a *heap type* from the *spec* (see `Py_TPFLAGS_HEAPTYPE`).

The metaclass *metaclass* is used to construct the resulting type object. When *metaclass* is `NULL`, the metaclass is derived from *bases* (or *Py_tp_base[s]* slots if *bases* is `NULL`, see below).

Metaclasses that override `tp_new` are not supported, except if `tp_new` is `NULL`. (For backwards compatibility, other `PyType_From*` functions allow such metaclasses. They ignore `tp_new`, which may result in incomplete initialization. This is deprecated and in Python 3.14+ such metaclasses will not be supported.)

The *bases* argument can be used to specify base classes; it can either be only one class or a tuple of classes. If *bases* is `NULL`, the *Py_tp_bases* slot is used instead. If that also is `NULL`, the *Py_tp_base* slot is used instead. If that also is `NULL`, the new type derives from `object`.

The *module* argument can be used to record the module in which the new class is defined. It must be a module object or `NULL`. If not `NULL`, the module is associated with the new type and can later be retrieved with `PyType_GetModule()`. The associated module is not inherited by subclasses; it must be specified for each class individually.

This function calls `PyType_Ready()` on the new type.

Note that this function does *not* fully match the behavior of calling `type()` or using the `class` statement. With user-provided base types or metaclasses, prefer *calling* `type` (or the metaclass) over `PyType_From*` functions. Specifically:

- `__new__()` is not called on the new class (and it must be set to `type.__new__`).
- `__init__()` is not called on the new class.
- `__init_subclass__()` is not called on any bases.
- `__set_name__()` is not called on new descriptors.

Added in version 3.12.

PyObject ***PyType_FromModuleAndSpec***(*PyObject* *module, *PyType_Spec* *spec, *PyObject* *bases)

Wartość zwracana: nowa referencja. Część stabilnego ABI *od wersji 3.10.* Equivalent to `PyType_FromMetaclass(NULL, module, spec, bases)`.

Added in version 3.9.

Zmienione w wersji 3.10: The function now accepts a single class as the *bases* argument and `NULL` as the `tp_doc` slot.

Zmienione w wersji 3.12: The function now finds and uses a metaclass corresponding to the provided base classes. Previously, only `type` instances were returned.

The `tp_new` of the metaclass is *ignored*. which may result in incomplete initialization. Creating classes whose metaclass overrides `tp_new` is deprecated and in Python 3.14+ it will be no longer allowed.

PyObject ***PyType_FromSpecWithBases***(*PyType_Spec* *spec, *PyObject* *bases)

Wartość zwracana: nowa referencja. Część stabilnego ABI *od wersji 3.3.* Equivalent to `PyType_FromMetaclass(NULL, NULL, spec, bases)`.

Added in version 3.3.

Zmienione w wersji 3.12: The function now finds and uses a metaclass corresponding to the provided base classes. Previously, only `type` instances were returned.

The `tp_new` of the metaclass is *ignored*. which may result in incomplete initialization. Creating classes whose metaclass overrides `tp_new` is deprecated and in Python 3.14+ it will be no longer allowed.

PyObject ***PyType_FromSpec***(*PyType_Spec* *spec)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Equivalent to `PyType_FromMetaclass(NULL, NULL, spec, NULL)`.

Zmienione w wersji 3.12: The function now finds and uses a metaclass corresponding to the base classes provided in *Py_tp_base[s]* slots. Previously, only `type` instances were returned.

The `tp_new` of the metaclass is *ignored*. which may result in incomplete initialization. Creating classes whose metaclass overrides `tp_new` is deprecated and in Python 3.14+ it will be no longer allowed.

type ***PyType_Spec***

Część stabilnego ABI *(w tym wszystkie składniki).* Structure defining a type's behavior.

const char ***name***

Name of the type, used to set `PyTypeObject.tp_name`.

int **basicsize**

If positive, specifies the size of the instance in bytes. It is used to set `PyTypeObject.tp_basicsize`.

If zero, specifies that `tp_basicsize` should be inherited.

If negative, the absolute value specifies how much space instances of the class need *in addition* to the superclass. Use `PyObject_GetTypeData()` to get a pointer to subclass-specific memory reserved this way.

Zmienione w wersji 3.12: Previously, this field could not be negative.

int **itemsize**

Size of one element of a variable-size type, in bytes. Used to set `PyTypeObject.tp_itemsize`. See `tp_itemsize` documentation for caveats.

If zero, `tp_itemsize` is inherited. Extending arbitrary variable-sized classes is dangerous, since some types use a fixed offset for variable-sized memory, which can then overlap fixed-sized memory used by a subclass. To help prevent mistakes, inheriting `itemsize` is only possible in the following situations:

- The base is not variable-sized (its `tp_itemsize`).
- The requested `PyType_Spec.basicsize` is positive, suggesting that the memory layout of the base class is known.
- The requested `PyType_Spec.basicsize` is zero, suggesting that the subclass does not access the instance's memory directly.
- With the `Py_TPFLAGS_ITEMS_AT_END` flag.

unsigned int **flags**

Type flags, used to set `PyTypeObject.tp_flags`.

If the `Py_TPFLAGS_HEAPTYPE` flag is not set, `PyType_FromSpecWithBases()` sets it automatically.

PyType_Slot ***slots**

Array of `PyType_Slot` structures. Terminated by the special slot value `{0, NULL}`.

Each slot ID should be specified at most once.

type **PyType_Slot**

Część stabilnego ABI *(w tym wszystkie składniki).* Structure defining optional functionality of a type, containing a slot ID and a value pointer.

int **slot**

A slot ID.

Slot IDs are named like the field names of the structures `PyTypeObject`, `PyNumberMethods`, `PySequenceMethods`, `PyMappingMethods` and `PyAsyncMethods` with an added `Py_` prefix. For example, use:

- `Py_tp_dealloc` to set `PyTypeObject.tp_dealloc`
- `Py_nb_add` to set `PyNumberMethods.nb_add`
- `Py_sq_length` to set `PySequenceMethods.sq_length`

The following "offset" fields cannot be set using `PyType_Slot`:

- `tp_weaklistoffset` (use `Py_TPFLAGS_MANAGED_WEAKREF` instead if possible)
- `tp_dictoffset` (use `Py_TPFLAGS_MANAGED_DICT` instead if possible)
- `tp_vectorcall_offset` (use `"__vectorcalloffset__"` in *PyMemberDef*)

If it is not possible to switch to a `MANAGED` flag (for example, for vectorcall or to support Python older than 3.12), specify the offset in `Py_tp_members`. See *PyMemberDef documentation* for details.

The following fields cannot be set at all when creating a heap type:

- `tp_vectorcall` (use `tp_new` and/or `tp_init`)
- Internal fields: `tp_dict`, `tp_mro`, `tp_cache`, `tp_subclasses`, and `tp_weaklist`.

Setting `Py_tp_bases` or `Py_tp_base` may be problematic on some platforms. To avoid issues, use the *bases* argument of `PyType_FromSpecWithBases()` instead.

Zmienione w wersji 3.9: Slots in `PyBufferProcs` may be set in the unlimited API.

Zmienione w wersji 3.11: `bf_getbuffer` and `bf_releasebuffer` are now available under the *limited API*.

void ***pfunc**

The desired value of the slot. In most cases, this is a pointer to a function.

Slots other than `Py_tp_doc` may not be `NULL`.

8.1.2 The `None` Object

Note that the `PyTypeObject` for `None` is not directly exposed in the Python/C API. Since `None` is a singleton, testing for object identity (using == in C) is sufficient. There is no `PyNone_Check()` function for the same reason.

PyObject ***Py_None**

The Python `None` object, denoting lack of value. This object has no methods and is *immortal*.

Zmienione w wersji 3.12: `Py_None` is *immortal*.

Py_RETURN_NONE

Return `Py_None` from a function.

8.2 Numeric Objects

8.2.1 Integer Objects

All integers are implemented as „long" integer objects of arbitrary size.

On error, most `PyLong_As*` APIs return `(return type)-1` which cannot be distinguished from a number. Use `PyErr_Occurred()` to disambiguate.

type **PyLongObject**

Część ograniczonego API *(jako nieprzezroczysta struktura).* This subtype of `PyObject` represents a Python integer object.

PyTypeObject **PyLong_Type**

Część stabilnego ABI. This instance of `PyTypeObject` represents the Python integer type. This is the same object as `int` in the Python layer.

int **PyLong_Check**(*PyObject* *p)

Return true if its argument is a `PyLongObject` or a subtype of `PyLongObject`. This function always succeeds.

int **PyLong_CheckExact**(*PyObject* *p)

Return true if its argument is a `PyLongObject`, but not a subtype of `PyLongObject`. This function always succeeds.

PyObject ***PyLong_FromLong**(long v)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return a new `PyLongObject` object from *v*, or `NULL` on failure.

The current implementation keeps an array of integer objects for all integers between `-5` and `256`. When you create an int in that range you actually just get back a reference to the existing object.

PyObject ***PyLong_FromUnsignedLong**(unsigned long v)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return a new `PyLongObject` object from a C `unsigned long`, or `NULL` on failure.

PyObject ***PyLong_FromSsize_t**(*Py_ssize_t* v)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return a new `PyLongObject` object from a C `Py_ssize_t`, or `NULL` on failure.

PyObject ***PyLong_FromSize_t**(size_t v)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return a new `PyLongObject` object from a C `size_t`, or `NULL` on failure.

PyObject ***PyLong_FromLongLong**(long long v)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return a new `PyLongObject` object from a C `long long`, or `NULL` on failure.

PyObject ***PyLong_FromUnsignedLongLong**(unsigned long long v)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return a new `PyLongObject` object from a C `unsigned long long`, or `NULL` on failure.

PyObject ***PyLong_FromDouble**(double v)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return a new `PyLongObject` object from the integer part of *v*, or `NULL` on failure.

PyObject ***PyLong_FromString**(const char *str, char **pend, int base)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return a new `PyLongObject` based on the string value in *str*, which is interpreted according to the radix in *base*, or `NULL` on failure. If *pend* is non-`NULL`, **pend* will point to the end of *str* on success or to the first character that could not be processed on error. If *base* is `0`, *str* is interpreted using the integers definition; in this case, leading zeros in a non-zero decimal number raises a `ValueError`. If *base* is not `0`, it must be between `2` and `36`, inclusive. Leading and trailing whitespace and single underscores after a base specifier and between digits are ignored. If there are no digits or *str* is not NULL-terminated following the digits and trailing whitespace, `ValueError` will be raised.

> **Zobacz także**
>
> Python methods `int.to_bytes()` and `int.from_bytes()` to convert a `PyLongObject` to/from an array of bytes in base `256`. You can call those from C using `PyObject_CallMethod()`.

PyObject ***PyLong_FromUnicodeObject**(*PyObject* *u, int base)

Wartość zwracana: nowa referencja. Convert a sequence of Unicode digits in the string *u* to a Python integer value.

Added in version 3.3.

PyObject ***PyLong_FromVoidPtr**(void *p)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Create a Python integer from the pointer *p*. The pointer value can be retrieved from the resulting value using `PyLong_AsVoidPtr()`.

PyObject ***PyLong_FromNativeBytes**(const void *buffer, size_t n_bytes, int flags)

Create a Python integer from the value contained in the first *n_bytes* of *buffer*, interpreted as a two's--complement signed number.

flags are as for `PyLong_AsNativeBytes()`. Passing `-1` will select the native endian that CPython was compiled with and assume that the most-significant bit is a sign bit. Passing `Py_ASNATIVEBYTES_UNSIGNED_BUFFER` will produce the same result as calling `PyLong_FromUnsignedNativeBytes()`. Other flags are ignored.

Added in version 3.13.

PyObject ***PyLong_FromUnsignedNativeBytes**(const void *buffer, size_t n_bytes, int flags)

Create a Python integer from the value contained in the first *n_bytes* of *buffer*, interpreted as an unsigned number.

flags are as for `PyLong_AsNativeBytes()`. Passing `-1` will select the native endian that CPython was compiled with and assume that the most-significant bit is not a sign bit. Flags other than endian are ignored.

Added in version 3.13.

long **PyLong_AsLong**(*PyObject* *obj)

Część stabilnego ABI. Return a C `long` representation of *obj*. If *obj* is not an instance of `PyLongObject`, first call its `__index__()` method (if present) to convert it to a `PyLongObject`.

Raise `OverflowError` if the value of *obj* is out of range for a `long`.

Returns `-1` on error. Use `PyErr_Occurred()` to disambiguate.

Zmienione w wersji 3.8: Use `__index__()` if available.

Zmienione w wersji 3.10: This function will no longer use `__int__()`.

long **PyLong_AS_LONG**(*PyObject* *obj)

A *soft deprecated* alias. Exactly equivalent to the preferred `PyLong_AsLong`. In particular, it can fail with `OverflowError` or another exception.

Niezalecane od wersji 3.14: The function is soft deprecated.

int **PyLong_AsInt**(*PyObject* *obj)

Część stabilnego ABI *od wersji 3.13*. Similar to `PyLong_AsLong()`, but store the result in a C `int` instead of a C `long`.

Added in version 3.13.

long **PyLong_AsLongAndOverflow**(*PyObject* *obj, int *overflow)

Część stabilnego ABI. Return a C `long` representation of *obj*. If *obj* is not an instance of `PyLongObject`, first call its `__index__()` method (if present) to convert it to a `PyLongObject`.

If the value of *obj* is greater than `LONG_MAX` or less than `LONG_MIN`, set **overflow* to `1` or `-1`, respectively, and return `-1`; otherwise, set **overflow* to `0`. If any other exception occurs set **overflow* to `0` and return `-1` as usual.

Returns `-1` on error. Use `PyErr_Occurred()` to disambiguate.

Zmienione w wersji 3.8: Use `__index__()` if available.

Zmienione w wersji 3.10: This function will no longer use `__int__()`.

long long **PyLong_AsLongLong**(*PyObject* *obj)

Część stabilnego ABI. Return a C `long long` representation of *obj*. If *obj* is not an instance of `PyLongObject`, first call its `__index__()` method (if present) to convert it to a `PyLongObject`.

Raise `OverflowError` if the value of *obj* is out of range for a `long long`.

Returns `-1` on error. Use `PyErr_Occurred()` to disambiguate.

Zmienione w wersji 3.8: Use `__index__()` if available.

Zmienione w wersji 3.10: This function will no longer use `__int__()`.

long long **PyLong_AsLongLongAndOverflow**(*PyObject* *obj, int *overflow)

Część stabilnego ABI. Return a C `long long` representation of *obj*. If *obj* is not an instance of `PyLongObject`, first call its `__index__()` method (if present) to convert it to a `PyLongObject`.

If the value of *obj* is greater than `LLONG_MAX` or less than `LLONG_MIN`, set **overflow* to `1` or `-1`, respectively, and return `-1`; otherwise, set **overflow* to `0`. If any other exception occurs set **overflow* to `0` and return `-1` as usual.

Returns `-1` on error. Use `PyErr_Occurred()` to disambiguate.

Added in version 3.2.

Zmienione w wersji 3.8: Use `__index__()` if available.

Zmienione w wersji 3.10: This function will no longer use `__int__()`.

Py_ssize_t **PyLong_AsSsize_t**(*PyObject* *pylong)

Część stabilnego ABI. Return a C `Py_ssize_t` representation of *pylong*. *pylong* must be an instance of `PyLongObject`.

Raise `OverflowError` if the value of *pylong* is out of range for a `Py_ssize_t`.

Returns `-1` on error. Use `PyErr_Occurred()` to disambiguate.

unsigned long **PyLong_AsUnsignedLong**(*PyObject* *pylong)

Część stabilnego ABI. Return a C `unsigned long` representation of *pylong*. *pylong* must be an instance of `PyLongObject`.

Raise `OverflowError` if the value of *pylong* is out of range for a `unsigned long`.

Returns `(unsigned long)-1` on error. Use `PyErr_Occurred()` to disambiguate.

size_t **PyLong_AsSize_t**(*PyObject* *pylong)

Część stabilnego ABI. Return a C `size_t` representation of *pylong*. *pylong* must be an instance of `PyLongObject`.

Raise `OverflowError` if the value of *pylong* is out of range for a `size_t`.

Returns `(size_t)-1` on error. Use `PyErr_Occurred()` to disambiguate.

unsigned long long **PyLong_AsUnsignedLongLong**(*PyObject* *pylong)

Część stabilnego ABI. Return a C `unsigned long long` representation of *pylong*. *pylong* must be an instance of `PyLongObject`.

Raise `OverflowError` if the value of *pylong* is out of range for an `unsigned long long`.

Returns `(unsigned long long)-1` on error. Use `PyErr_Occurred()` to disambiguate.

Zmienione w wersji 3.1: A negative *pylong* now raises `OverflowError`, not `TypeError`.

unsigned long **PyLong_AsUnsignedLongMask**(*PyObject* *obj)

Część stabilnego ABI. Return a C `unsigned long` representation of *obj*. If *obj* is not an instance of `PyLongObject`, first call its `__index__()` method (if present) to convert it to a `PyLongObject`.

If the value of *obj* is out of range for an `unsigned long`, return the reduction of that value modulo `ULONG_MAX + 1`.

Returns `(unsigned long)-1` on error. Use `PyErr_Occurred()` to disambiguate.

Zmienione w wersji 3.8: Use `__index__()` if available.

Zmienione w wersji 3.10: This function will no longer use `__int__()`.

unsigned long long **PyLong_AsUnsignedLongLongMask**(*PyObject* *obj)

Część stabilnego ABI. Return a C `unsigned long long` representation of *obj*. If *obj* is not an instance of `PyLongObject`, first call its `__index__()` method (if present) to convert it to a `PyLongObject`.

If the value of *obj* is out of range for an `unsigned long long`, return the reduction of that value modulo `ULLONG_MAX + 1`.

Returns `(unsigned long long)-1` on error. Use `PyErr_Occurred()` to disambiguate.

Zmienione w wersji 3.8: Use `__index__()` if available.

Zmienione w wersji 3.10: This function will no longer use `__int__()`.

double **PyLong_AsDouble**(*PyObject* *pylong)

Część stabilnego ABI. Return a C `double` representation of *pylong*. *pylong* must be an instance of `PyLongObject`.

Raise `OverflowError` if the value of *pylong* is out of range for a `double`.

Returns `-1.0` on error. Use `PyErr_Occurred()` to disambiguate.

void ***PyLong_AsVoidPtr**(*PyObject* *pylong)

Część stabilnego ABI. Convert a Python integer *pylong* to a C `void` pointer. If *pylong* cannot be converted, an `OverflowError` will be raised. This is only assured to produce a usable `void` pointer for values created with `PyLong_FromVoidPtr()`.

Returns `NULL` on error. Use `PyErr_Occurred()` to disambiguate.

Py_ssize_t **PyLong_AsNativeBytes**(*PyObject* *pylong, void *buffer, *Py_ssize_t* n_bytes, int flags)

Copy the Python integer value *pylong* to a native *buffer* of size *n_bytes*. The *flags* can be set to `-1` to behave similarly to a C cast, or to values documented below to control the behavior.

Returns `-1` with an exception raised on error. This may happen if *pylong* cannot be interpreted as an integer, or if *pylong* was negative and the `Py_ASNATIVEBYTES_REJECT_NEGATIVE` flag was set.

Otherwise, returns the number of bytes required to store the value. If this is equal to or less than *n_bytes*, the entire value was copied. All *n_bytes* of the buffer are written: large buffers are padded with zeroes.

If the returned value is greater than than *n_bytes*, the value was truncated: as many of the lowest bits of the value as could fit are written, and the higher bits are ignored. This matches the typical behavior of a C-style downcast.

> **Informacja**
>
> Overflow is not considered an error. If the returned value is larger than *n_bytes*, most significant bits were discarded.

`0` will never be returned.

Values are always copied as two's-complement.

Usage example:

```
int32_t value;
Py_ssize_t bytes = PyLong_AsNativeBytes(pylong, &value, sizeof(value), -1);
if (bytes < 0) {
    // Failed. A Python exception was set with the reason.
    return NULL;
}
else if (bytes <= (Py_ssize_t)sizeof(value)) {
    // Success!
}
else {
    // Overflow occurred, but 'value' contains the truncated
    // lowest bits of pylong.
}
```

Passing zero to *n_bytes* will return the size of a buffer that would be large enough to hold the value. This may be larger than technically necessary, but not unreasonably so. If *n_bytes=0*, *buffer* may be `NULL`.

> **Informacja**
>
> Passing *n_bytes=0* to this function is not an accurate way to determine the bit length of the value.

To get at the entire Python value of an unknown size, the function can be called twice: first to determine the buffer size, then to fill it:

```
// Ask how much space we need.
Py_ssize_t expected = PyLong_AsNativeBytes(pylong, NULL, 0, -1);
if (expected < 0) {
    // Failed. A Python exception was set with the reason.
    return NULL;
}
assert(expected != 0);  // Impossible per the API definition.
uint8_t *bignum = malloc(expected);
if (!bignum) {
    PyErr_SetString(PyExc_MemoryError, "bignum malloc failed.");
    return NULL;
}
// Safely get the entire value.
Py_ssize_t bytes = PyLong_AsNativeBytes(pylong, bignum, expected, -1);
if (bytes < 0) {  // Exception has been set.
    free(bignum);
    return NULL;
}
else if (bytes > expected) {  // This should not be possible.
    PyErr_SetString(PyExc_RuntimeError,
        "Unexpected bignum truncation after a size check.");
    free(bignum);
    return NULL;
}
// The expected success given the above pre-check.
// ... use bignum ...
free(bignum);
```

flags is either `-1` (`Py_ASNATIVEBYTES_DEFAULTS`) to select defaults that behave most like a C cast, or a combintation of the other flags in the table below. Note that `-1` cannot be combined with other flags.

Currently, `-1` corresponds to `Py_ASNATIVEBYTES_NATIVE_ENDIAN | Py_ASNATIVEBYTES_UNSIGNED_BUFFER`.

Flag	Wartość
Py_ASNATIVEBYTES_DEFAULTS	-1
Py_ASNATIVEBYTES_BIG_ENDIAN	0
Py_ASNATIVEBYTES_LITTLE_ENDIAN	1
Py_ASNATIVEBYTES_NATIVE_ENDIAN	3
Py_ASNATIVEBYTES_UNSIGNED_BUFFER	4
Py_ASNATIVEBYTES_REJECT_NEGATIVE	8
Py_ASNATIVEBYTES_ALLOW_INDEX	16

Specifying `Py_ASNATIVEBYTES_NATIVE_ENDIAN` will override any other endian flags. Passing `2` is reserved.

By default, sufficient buffer will be requested to include a sign bit. For example, when converting 128 with *n_bytes=1*, the function will return 2 (or more) in order to store a zero sign bit.

If `Py_ASNATIVEBYTES_UNSIGNED_BUFFER` is specified, a zero sign bit will be omitted from size calculations. This allows, for example, 128 to fit in a single-byte buffer. If the destination buffer is later treated as signed, a positive input value may become negative. Note that the flag does not affect handling of negative values: for those, space for a sign bit is always requested.

Specifying `Py_ASNATIVEBYTES_REJECT_NEGATIVE` causes an exception to be set if *pylong* is negative. Without this flag, negative values will be copied provided there is enough space for at least one sign bit, regardless of whether `Py_ASNATIVEBYTES_UNSIGNED_BUFFER` was specified.

If `Py_ASNATIVEBYTES_ALLOW_INDEX` is specified and a non-integer value is passed, its `__index__()` method will be called first. This may result in Python code executing and other threads being allowed to run, which could cause changes to other objects or values in use. When *flags* is `-1`, this option is not set, and non-integer values will raise `TypeError`.

> **Informacja**
>
> With the default *flags* (`-1`, or *UNSIGNED_BUFFER* without *REJECT_NEGATIVE*), multiple Python integers can map to a single value without overflow. For example, both `255` and `-1` fit a single-byte buffer and set all its bits. This matches typical C cast behavior.

Added in version 3.13.

PyObject ***PyLong_GetInfo**(void)

Część stabilnego ABI. On success, return a read only *named tuple*, that holds information about Python's internal representation of integers. See `sys.int_info` for description of individual fields.

On failure, return `NULL` with an exception set.

Added in version 3.1.

int **PyUnstable_Long_IsCompact**(const *PyLongObject* *op)

> To jest *niestabilne API*. Może ulec zmianie bez ostrzeżenia w wydaniach minor.

Return 1 if *op* is compact, 0 otherwise.

This function makes it possible for performance-critical code to implement a "fast path" for small integers. For compact values use `PyUnstable_Long_CompactValue()`; for others fall back to a `PyLong_As*` function or `PyLong_AsNativeBytes()`.

The speedup is expected to be negligible for most users.

Exactly what values are considered compact is an implementation detail and is subject to change.

Added in version 3.12.

Py_ssize_t **PyUnstable_Long_CompactValue**(const *PyLongObject* *op)

> To jest *niestabilne API*. Może ulec zmianie bez ostrzeżenia w wydaniach minor.

If *op* is compact, as determined by `PyUnstable_Long_IsCompact()`, return its value.

Otherwise, the return value is undefined.

Added in version 3.12.

8.2.2 Obiekty logiczne

Booleans in Python are implemented as a subclass of integers. There are only two booleans, `Py_False` and `Py_True`. As such, the normal creation and deletion functions don't apply to booleans. The following macros are available, however.

PyTypeObject **PyBool_Type**

Część stabilnego ABI. This instance of `PyTypeObject` represents the Python boolean type; it is the same object as `bool` in the Python layer.

int **PyBool_Check**(*PyObject* *o)

Zwraca wartość true, jeśli *o* jest typu `PyBool_Type`. Ta funkcja zawsze kończy się powodzeniem.

PyObject ***Py_False**

The Python `False` object. This object has no methods and is *immortal*.

Zmienione w wersji 3.12: `Py_False` is *immortal*.

PyObject ***Py_True**

The Python `True` object. This object has no methods and is *immortal*.

Zmienione w wersji 3.12: `Py_True` is *immortal*.

Py_RETURN_FALSE

Return `Py_False` from a function.

Py_RETURN_TRUE

Return `Py_True` from a function.

PyObject ***PyBool_FromLong**(long v)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return `Py_True` or `Py_False`, depending on the truth value of *v*.

8.2.3 Floating-Point Objects

type **PyFloatObject**

This subtype of `PyObject` represents a Python floating-point object.

PyTypeObject **PyFloat_Type**

Część stabilnego ABI. This instance of `PyTypeObject` represents the Python floating-point type. This is the same object as `float` in the Python layer.

int **PyFloat_Check**(*PyObject* *p)

Return true if its argument is a `PyFloatObject` or a subtype of `PyFloatObject`. This function always succeeds.

int **PyFloat_CheckExact**(*PyObject* *p)

Return true if its argument is a `PyFloatObject`, but not a subtype of `PyFloatObject`. This function always succeeds.

PyObject ***PyFloat_FromString**(*PyObject* *str)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Create a `PyFloatObject` object based on the string value in *str*, or `NULL` on failure.

PyObject ***PyFloat_FromDouble**(double v)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Create a `PyFloatObject` object from *v*, or `NULL` on failure.

double **PyFloat_AsDouble**(*PyObject* *pyfloat)

Część stabilnego ABI. Return a C `double` representation of the contents of *pyfloat*. If *pyfloat* is not a Python floating-point object but has a `__float__()` method, this method will first be called to convert *pyfloat* into a float. If `__float__()` is not defined then it falls back to `__index__()`. This method returns `-1.0` upon failure, so one should call `PyErr_Occurred()` to check for errors.

Zmienione w wersji 3.8: Use `__index__()` if available.

double **PyFloat_AS_DOUBLE**(*PyObject* *pyfloat)

Return a C `double` representation of the contents of *pyfloat*, but without error checking.

PyObject ***PyFloat_GetInfo**(void)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return a structseq instance which contains information about the precision, minimum and maximum values of a float. It's a thin wrapper around the header file `float.h`.

double **PyFloat_GetMax**()

Część stabilnego ABI. Return the maximum representable finite float *DBL_MAX* as C `double`.

double **PyFloat_GetMin**()

Część stabilnego ABI. Return the minimum normalized positive float *DBL_MIN* as C `double`.

Pack and Unpack functions

The pack and unpack functions provide an efficient platform-independent way to store floating-point values as byte strings. The Pack routines produce a bytes string from a C `double`, and the Unpack routines produce a C `double` from such a bytes string. The suffix (2, 4 or 8) specifies the number of bytes in the bytes string.

On platforms that appear to use IEEE 754 formats these functions work by copying bits. On other platforms, the 2-byte format is identical to the IEEE 754 binary16 half-precision format, the 4-byte format (32-bit) is identical to the IEEE 754 binary32 single precision format, and the 8-byte format to the IEEE 754 binary64 double precision format, although the packing of INFs and NaNs (if such things exist on the platform) isn't handled correctly, and attempting to unpack a bytes string containing an IEEE INF or NaN will raise an exception.

On non-IEEE platforms with more precision, or larger dynamic range, than IEEE 754 supports, not all values can be packed; on non-IEEE platforms with less precision, or smaller dynamic range, not all values can be unpacked. What happens in such cases is partly accidental (alas).

Added in version 3.11.

Pack functions

The pack routines write 2, 4 or 8 bytes, starting at *p*. *le* is an `int` argument, non-zero if you want the bytes string in little-endian format (exponent last, at `p+1`, `p+3`, or `p+6` `p+7`), zero if you want big-endian format (exponent first, at *p*). The `PY_BIG_ENDIAN` constant can be used to use the native endian: it is equal to `1` on big endian processor, or `0` on little endian processor.

Return value: `0` if all is OK, `-1` if error (and an exception is set, most likely `OverflowError`).

There are two problems on non-IEEE platforms:

- What this does is undefined if *x* is a NaN or infinity.
- `-0.0` and `+0.0` produce the same bytes string.

int **PyFloat_Pack2**(double x, unsigned char *p, int le)

Pack a C double as the IEEE 754 binary16 half-precision format.

int **PyFloat_Pack4**(double x, unsigned char *p, int le)

Pack a C double as the IEEE 754 binary32 single precision format.

int **PyFloat_Pack8**(double x, unsigned char *p, int le)

Pack a C double as the IEEE 754 binary64 double precision format.

Unpack functions

The unpack routines read 2, 4 or 8 bytes, starting at *p*. *le* is an `int` argument, non-zero if the bytes string is in little-endian format (exponent last, at `p+1`, `p+3` or `p+6` and `p+7`), zero if big-endian (exponent first, at *p*). The `PY_BIG_ENDIAN` constant can be used to use the native endian: it is equal to `1` on big endian processor, or `0` on little endian processor.

Return value: The unpacked double. On error, this is `-1.0` and *`PyErr_Occurred()`* is true (and an exception is set, most likely `OverflowError`).

Note that on a non-IEEE platform this will refuse to unpack a bytes string that represents a NaN or infinity.

double **PyFloat_Unpack2**(const unsigned char *p, int le)

Unpack the IEEE 754 binary16 half-precision format as a C double.

double **PyFloat_Unpack4**(const unsigned char *p, int le)

Unpack the IEEE 754 binary32 single precision format as a C double.

double **PyFloat_Unpack8**(const unsigned char *p, int le)

Unpack the IEEE 754 binary64 double precision format as a C double.

8.2.4 Obiekt Liczby Zespolonej

Python-owe obiekty liczby zespolonej są stworzone w C jako dwa oddzielne typy: jeden jest strukturą w która reprezentuje prawdziwe liczby zespolone, a drugi ujawnia tą strukturę dla Python-owego kodu. API ma funkcje do operacji na obydwu typach.

Liczby zespolone jako struktury w C

Note that the functions which accept these structures as parameters and return them as results do so *by value* rather than dereferencing them through pointers. This is consistent throughout the API.

type **Py_complex**

The C structure which corresponds to the value portion of a Python complex number object. Most of the functions for dealing with complex number objects use structures of this type as input or output values, as appropriate.

double **real**

double **imag**

The structure is defined as:

```
typedef struct {
    double real;
    double imag;
} Py_complex;
```

Py_complex **_Py_c_sum**(*Py_complex* left, *Py_complex* right)

Zwraca sumę dwóch liczb zespolonych wykorzystując strukturę `Py_complex` jako reprezentację.

Py_complex **_Py_c_diff**(*Py_complex* left, *Py_complex* right)

Zwraca różnicę dwóch liczb zespolonych wykorzystując strukturę `Py_complex` jako reprezentację.

Py_complex **_Py_c_neg**(*Py_complex* num)

Return the negation of the complex number *num*, using the C `Py_complex` representation.

Py_complex **_Py_c_prod**(*Py_complex* left, *Py_complex* right)

Zwraca iloczyn dwóch liczb zespolonych wykorzystując strukturę `Py_complex` jako reprezentację.

Py_complex **_Py_c_quot**(*Py_complex* dividend, *Py_complex* divisor)

Zwraca iloraz dwóch liczb zespolonych wykorzystując strukturę `Py_complex` jako reprezentację.

If *divisor* is null, this method returns zero and sets `errno` to `EDOM`.

Py_complex **_Py_c_pow**(*Py_complex* num, *Py_complex* exp)

Return the exponentiation of *num* by *exp*, using the C `Py_complex` representation.

If *num* is null and *exp* is not a positive real number, this method returns zero and sets `errno` to `EDOM`.

Complex Numbers as Python Objects

type **PyComplexObject**

This subtype of `PyObject` represents a Python complex number object.

PyTypeObject **PyComplex_Type**

Część stabilnego ABI. This instance of `PyTypeObject` represents the Python complex number type. It is the same object as `complex` in the Python layer.

int **PyComplex_Check**(*PyObject* *p)

Return true if its argument is a `PyComplexObject` or a subtype of `PyComplexObject`. This function always succeeds.

int **PyComplex_CheckExact**(*PyObject* *p)

Return true if its argument is a `PyComplexObject`, but not a subtype of `PyComplexObject`. This function always succeeds.

PyObject ***PyComplex_FromCComplex**(*Py_complex* v)

Wartość zwracana: nowa referencja. Create a new Python complex number object from a C `Py_complex` value. Return `NULL` with an exception set on error.

PyObject ***PyComplex_FromDoubles**(double real, double imag)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return a new `PyComplexObject` object from *real* and *imag*. Return `NULL` with an exception set on error.

double **PyComplex_RealAsDouble**(*PyObject* *op)

Część stabilnego ABI. Return the real part of *op* as a C `double`.

If *op* is not a Python complex number object but has a `__complex__()` method, this method will first be called to convert *op* to a Python complex number object. If `__complex__()` is not defined then it falls back to call `PyFloat_AsDouble()` and returns its result.

Upon failure, this method returns `-1.0` with an exception set, so one should call `PyErr_Occurred()` to check for errors.

Zmienione w wersji 3.13: Use `__complex__()` if available.

double **PyComplex_ImagAsDouble**(*PyObject* *op)

Część stabilnego ABI. Return the imaginary part of *op* as a C `double`.

If *op* is not a Python complex number object but has a `__complex__()` method, this method will first be called to convert *op* to a Python complex number object. If `__complex__()` is not defined then it falls back to call `PyFloat_AsDouble()` and returns `0.0` on success.

Upon failure, this method returns `-1.0` with an exception set, so one should call `PyErr_Occurred()` to check for errors.

Zmienione w wersji 3.13: Use `__complex__()` if available.

Py_complex **PyComplex_AsCComplex**(*PyObject* *op)

Return the `Py_complex` value of the complex number *op*.

If *op* is not a Python complex number object but has a `__complex__()` method, this method will first be called to convert *op* to a Python complex number object. If `__complex__()` is not defined then it falls back to `__float__()`. If `__float__()` is not defined then it falls back to `__index__()`.

Upon failure, this method returns `Py_complex` with `real` set to `-1.0` and with an exception set, so one should call `PyErr_Occurred()` to check for errors.

Zmienione w wersji 3.8: Use `__index__()` if available.

8.3 Sequence Objects

Generic operations on sequence objects were discussed in the previous chapter; this section deals with the specific kinds of sequence objects that are intrinsic to the Python language.

8.3.1 Bytes Objects

These functions raise `TypeError` when expecting a bytes parameter and called with a non-bytes parameter.

type **PyBytesObject**

This subtype of `PyObject` represents a Python bytes object.

PyTypeObject **PyBytes_Type**

Część stabilnego ABI. This instance of `PyTypeObject` represents the Python bytes type; it is the same object as `bytes` in the Python layer.

int **PyBytes_Check**(*PyObject* *o)

Return true if the object *o* is a bytes object or an instance of a subtype of the bytes type. This function always succeeds.

int **PyBytes_CheckExact**(*PyObject* *o)

Return true if the object *o* is a bytes object, but not an instance of a subtype of the bytes type. This function always succeeds.

PyObject ***PyBytes_FromString**(const char *v)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return a new bytes object with a copy of the string *v* as value on success, and `NULL` on failure. The parameter *v* must not be `NULL`; it will not be checked.

PyObject ***PyBytes_FromStringAndSize**(const char *v, *Py_ssize_t* len)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return a new bytes object with a copy of the string *v* as value and length *len* on success, and `NULL` on failure. If *v* is `NULL`, the contents of the bytes object are uninitialized.

PyObject ***PyBytes_FromFormat**(const char *format, ...)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Take a C `printf()`-style *format* string and a variable number of arguments, calculate the size of the resulting Python bytes object and return a bytes object with the values formatted into it. The variable arguments must be C types and must correspond exactly to the format characters in the *format* string. The following format characters are allowed:

Format Characters	Type	Comment
`%%`	*n/a*	The literal % character.
`%c`	int	A single byte, represented as a C int.
`%d`	int	Equivalent to `printf("%d")`.[1]
`%u`	nieoznaczony typ int	Equivalent to `printf("%u")`.[1]
`%ld`	long	Equivalent to `printf("%ld")`.[1]
`%lu`	nieoznaczony typ długi	Equivalent to `printf("%lu")`.[1]
`%zd`	`Py_ssize_t`	Equivalent to `printf("%zd")`.[1]
`%zu`	size_t	Equivalent to `printf("%zu")`.[1]
`%i`	int	Equivalent to `printf("%i")`.[1]
`%x`	int	Equivalent to `printf("%x")`.[1]
`%s`	const char*	A null-terminated C character array.
`%p`	const void*	The hex representation of a C pointer. Mostly equivalent to `printf("%p")` except that it is guaranteed to start with the literal `0x` regardless of what the platform's `printf` yields.

An unrecognized format character causes all the rest of the format string to be copied as-is to the result object, and any extra arguments discarded.

PyObject ***PyBytes_FromFormatV**(const char *format, va_list vargs)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Identical to `PyBytes_FromFormat()` except that it takes exactly two arguments.

PyObject ***PyBytes_FromObject**(*PyObject* *o)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return the bytes representation of object *o* that implements the buffer protocol.

Py_ssize_t **PyBytes_Size**(*PyObject* *o)

Część stabilnego ABI. Return the length of the bytes in bytes object *o*.

Py_ssize_t **PyBytes_GET_SIZE**(*PyObject* *o)

Similar to `PyBytes_Size()`, but without error checking.

char ***PyBytes_AsString**(*PyObject* *o)

Część stabilnego ABI. Return a pointer to the contents of *o*. The pointer refers to the internal buffer of *o*, which consists of `len(o) + 1` bytes. The last byte in the buffer is always null, regardless of whether there are any other null bytes. The data must not be modified in any way, unless the object was just created using `PyBytes_FromStringAndSize(NULL, size)`. It must not be deallocated. If *o* is not a bytes object at all, `PyBytes_AsString()` returns `NULL` and raises `TypeError`.

char ***PyBytes_AS_STRING**(*PyObject* *string)

Similar to `PyBytes_AsString()`, but without error checking.

int **PyBytes_AsStringAndSize**(*PyObject* *obj, char **buffer, *Py_ssize_t* *length)

Część stabilnego ABI. Return the null-terminated contents of the object *obj* through the output variables *buffer* and *length*. Returns `0` on success.

If *length* is `NULL`, the bytes object may not contain embedded null bytes; if it does, the function returns `-1` and a `ValueError` is raised.

[1] For integer specifiers (d, u, ld, lu, zd, zu, i, x): the 0-conversion flag has effect even when a precision is given.

The buffer refers to an internal buffer of *obj*, which includes an additional null byte at the end (not counted in *length*). The data must not be modified in any way, unless the object was just created using `PyBytes_FromStringAndSize(NULL, size)`. It must not be deallocated. If *obj* is not a bytes object at all, `PyBytes_AsStringAndSize()` returns `-1` and raises `TypeError`.

Zmienione w wersji 3.5: Previously, `TypeError` was raised when embedded null bytes were encountered in the bytes object.

void **PyBytes_Concat**(*PyObject* **bytes, *PyObject* *newpart)

Część stabilnego ABI. Create a new bytes object in **bytes* containing the contents of *newpart* appended to *bytes*; the caller will own the new reference. The reference to the old value of *bytes* will be stolen. If the new object cannot be created, the old reference to *bytes* will still be discarded and the value of **bytes* will be set to `NULL`; the appropriate exception will be set.

void **PyBytes_ConcatAndDel**(*PyObject* **bytes, *PyObject* *newpart)

Część stabilnego ABI. Create a new bytes object in **bytes* containing the contents of *newpart* appended to *bytes*. This version releases the *strong reference* to *newpart* (i.e. decrements its reference count).

int **_PyBytes_Resize**(*PyObject* **bytes, *Py_ssize_t* newsize)

Resize a bytes object. *newsize* will be the new length of the bytes object. You can think of it as creating a new bytes object and destroying the old one, only more efficiently. Pass the address of an existing bytes object as an lvalue (it may be written into), and the new size desired. On success, **bytes* holds the resized bytes object and `0` is returned; the address in **bytes* may differ from its input value. If the reallocation fails, the original bytes object at **bytes* is deallocated, **bytes* is set to `NULL`, `MemoryError` is set, and `-1` is returned.

8.3.2 Byte Array Objects

type **PyByteArrayObject**

This subtype of `PyObject` represents a Python bytearray object.

PyTypeObject **PyByteArray_Type**

Część stabilnego ABI. This instance of `PyTypeObject` represents the Python bytearray type; it is the same object as `bytearray` in the Python layer.

Type check macros

int **PyByteArray_Check**(*PyObject* *o)

Return true if the object *o* is a bytearray object or an instance of a subtype of the bytearray type. This function always succeeds.

int **PyByteArray_CheckExact**(*PyObject* *o)

Return true if the object *o* is a bytearray object, but not an instance of a subtype of the bytearray type. This function always succeeds.

Direct API functions

PyObject ***PyByteArray_FromObject**(*PyObject* *o)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return a new bytearray object from any object, *o*, that implements the *buffer protocol*.

On failure, return `NULL` with an exception set.

PyObject ***PyByteArray_FromStringAndSize**(const char *string, *Py_ssize_t* len)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Create a new bytearray object from *string* and its length, *len*.

On failure, return `NULL` with an exception set.

PyObject ***PyByteArray_Concat**(*PyObject* *a, *PyObject* *b)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Concat bytearrays *a* and *b* and return a new bytearray with the result.

On failure, return `NULL` with an exception set.

Py_ssize_t **PyByteArray_Size**(*PyObject* *bytearray)

Część stabilnego ABI. Return the size of *bytearray* after checking for a `NULL` pointer.

char ***PyByteArray_AsString**(*PyObject* *bytearray)

Część stabilnego ABI. Return the contents of *bytearray* as a char array after checking for a `NULL` pointer. The returned array always has an extra null byte appended.

int **PyByteArray_Resize**(*PyObject* *bytearray, *Py_ssize_t* len)

Część stabilnego ABI. Resize the internal buffer of *bytearray* to *len*.

Macros

These macros trade safety for speed and they don't check pointers.

char ***PyByteArray_AS_STRING**(*PyObject* *bytearray)

Similar to `PyByteArray_AsString()`, but without error checking.

Py_ssize_t **PyByteArray_GET_SIZE**(*PyObject* *bytearray)

Similar to `PyByteArray_Size()`, but without error checking.

8.3.3 Unicode Objects and Codecs

Unicode Objects

Since the implementation of **PEP 393** in Python 3.3, Unicode objects internally use a variety of representations, in order to allow handling the complete range of Unicode characters while staying memory efficient. There are special cases for strings where all code points are below 128, 256, or 65536; otherwise, code points must be below 1114112 (which is the full Unicode range).

UTF-8 representation is created on demand and cached in the Unicode object.

> **Informacja**
>
> The `Py_UNICODE` representation has been removed since Python 3.12 with deprecated APIs. See **PEP 623** for more information.

Unicode Type

These are the basic Unicode object types used for the Unicode implementation in Python:

type **Py_UCS4**

type **Py_UCS2**

type **Py_UCS1**

Część stabilnego ABI. These types are typedefs for unsigned integer types wide enough to contain characters of 32 bits, 16 bits and 8 bits, respectively. When dealing with single Unicode characters, use `Py_UCS4`.

Added in version 3.3.

type **Py_UNICODE**

This is a typedef of `wchar_t`, which is a 16-bit type or 32-bit type depending on the platform.

Zmienione w wersji 3.3: In previous versions, this was a 16-bit type or a 32-bit type depending on whether you selected a „narrow" or „wide" Unicode version of Python at build time.

Deprecated since version 3.13, will be removed in version 3.15.

type **PyASCIIObject**

type **PyCompactUnicodeObject**

type **PyUnicodeObject**

These subtypes of `PyObject` represent a Python Unicode object. In almost all cases, they shouldn't be used directly, since all API functions that deal with Unicode objects take and return `PyObject` pointers.

Added in version 3.3.

PyTypeObject **PyUnicode_Type**

Część stabilnego ABI. This instance of `PyTypeObject` represents the Python Unicode type. It is exposed to Python code as `str`.

The following APIs are C macros and static inlined functions for fast checks and access to internal read-only data of Unicode objects:

int **PyUnicode_Check**(*PyObject* *obj)

Return true if the object *obj* is a Unicode object or an instance of a Unicode subtype. This function always succeeds.

int **PyUnicode_CheckExact**(*PyObject* *obj)

Return true if the object *obj* is a Unicode object, but not an instance of a subtype. This function always succeeds.

int **PyUnicode_READY**(*PyObject* *unicode)

Returns `0`. This API is kept only for backward compatibility.

Added in version 3.3.

Niezalecane od wersji 3.10: This API does nothing since Python 3.12.

Py_ssize_t **PyUnicode_GET_LENGTH**(*PyObject* *unicode)

Return the length of the Unicode string, in code points. *unicode* has to be a Unicode object in the „canonical" representation (not checked).

Added in version 3.3.

Py_UCS1 ***PyUnicode_1BYTE_DATA**(*PyObject* *unicode)

Py_UCS2 ***PyUnicode_2BYTE_DATA**(*PyObject* *unicode)

Py_UCS4 ***PyUnicode_4BYTE_DATA**(*PyObject* *unicode)

Return a pointer to the canonical representation cast to UCS1, UCS2 or UCS4 integer types for direct character access. No checks are performed if the canonical representation has the correct character size; use `PyUnicode_KIND()` to select the right function.

Added in version 3.3.

PyUnicode_1BYTE_KIND

PyUnicode_2BYTE_KIND

PyUnicode_4BYTE_KIND

Return values of the `PyUnicode_KIND()` macro.

Added in version 3.3.

Zmienione w wersji 3.12: `PyUnicode_WCHAR_KIND` has been removed.

int **PyUnicode_KIND**(*PyObject* *unicode)

Return one of the PyUnicode kind constants (see above) that indicate how many bytes per character this Unicode object uses to store its data. *unicode* has to be a Unicode object in the „canonical" representation (not checked).

Added in version 3.3.

void *PyUnicode_DATA(PyObject *unicode)

Return a void pointer to the raw Unicode buffer. *unicode* has to be a Unicode object in the „canonical" representation (not checked).

Added in version 3.3.

void PyUnicode_WRITE(int kind, void *data, Py_ssize_t index, Py_UCS4 value)

Write into a canonical representation *data* (as obtained with `PyUnicode_DATA()`). This function performs no sanity checks, and is intended for usage in loops. The caller should cache the *kind* value and *data* pointer as obtained from other calls. *index* is the index in the string (starts at 0) and *value* is the new code point value which should be written to that location.

Added in version 3.3.

Py_UCS4 PyUnicode_READ(int kind, void *data, Py_ssize_t index)

Read a code point from a canonical representation *data* (as obtained with `PyUnicode_DATA()`). No checks or ready calls are performed.

Added in version 3.3.

Py_UCS4 PyUnicode_READ_CHAR(PyObject *unicode, Py_ssize_t index)

Read a character from a Unicode object *unicode*, which must be in the „canonical" representation. This is less efficient than `PyUnicode_READ()` if you do multiple consecutive reads.

Added in version 3.3.

Py_UCS4 PyUnicode_MAX_CHAR_VALUE(PyObject *unicode)

Return the maximum code point that is suitable for creating another string based on *unicode*, which must be in the „canonical" representation. This is always an approximation but more efficient than iterating over the string.

Added in version 3.3.

int PyUnicode_IsIdentifier(PyObject *unicode)

Część stabilnego ABI. Return `1` if the string is a valid identifier according to the language definition, section identifiers. Return `0` otherwise.

Zmienione w wersji 3.9: The function does not call `Py_FatalError()` anymore if the string is not ready.

Unicode Character Properties

Unicode provides many different character properties. The most often needed ones are available through these macros which are mapped to C functions depending on the Python configuration.

int Py_UNICODE_ISSPACE(Py_UCS4 ch)

Return `1` or `0` depending on whether *ch* is a whitespace character.

int Py_UNICODE_ISLOWER(Py_UCS4 ch)

Return `1` or `0` depending on whether *ch* is a lowercase character.

int Py_UNICODE_ISUPPER(Py_UCS4 ch)

Return `1` or `0` depending on whether *ch* is an uppercase character.

int Py_UNICODE_ISTITLE(Py_UCS4 ch)

Return `1` or `0` depending on whether *ch* is a titlecase character.

int Py_UNICODE_ISLINEBREAK(Py_UCS4 ch)

Return `1` or `0` depending on whether *ch* is a linebreak character.

int Py_UNICODE_ISDECIMAL(Py_UCS4 ch)

Return `1` or `0` depending on whether *ch* is a decimal character.

int Py_UNICODE_ISDIGIT(Py_UCS4 ch)

Return `1` or `0` depending on whether *ch* is a digit character.

int **Py_UNICODE_ISNUMERIC**(*Py_UCS4* ch)

Return `1` or `0` depending on whether *ch* is a numeric character.

int **Py_UNICODE_ISALPHA**(*Py_UCS4* ch)

Return `1` or `0` depending on whether *ch* is an alphabetic character.

int **Py_UNICODE_ISALNUM**(*Py_UCS4* ch)

Return `1` or `0` depending on whether *ch* is an alphanumeric character.

int **Py_UNICODE_ISPRINTABLE**(*Py_UCS4* ch)

Return `1` or `0` depending on whether *ch* is a printable character. Nonprintable characters are those characters defined in the Unicode character database as „Other" or „Separator", excepting the ASCII space (0x20) which is considered printable. (Note that printable characters in this context are those which should not be escaped when `repr()` is invoked on a string. It has no bearing on the handling of strings written to `sys.stdout` or `sys.stderr`.)

These APIs can be used for fast direct character conversions:

Py_UCS4 **Py_UNICODE_TOLOWER**(*Py_UCS4* ch)

Return the character *ch* converted to lower case.

Py_UCS4 **Py_UNICODE_TOUPPER**(*Py_UCS4* ch)

Return the character *ch* converted to upper case.

Py_UCS4 **Py_UNICODE_TOTITLE**(*Py_UCS4* ch)

Return the character *ch* converted to title case.

int **Py_UNICODE_TODECIMAL**(*Py_UCS4* ch)

Return the character *ch* converted to a decimal positive integer. Return `-1` if this is not possible. This function does not raise exceptions.

int **Py_UNICODE_TODIGIT**(*Py_UCS4* ch)

Return the character *ch* converted to a single digit integer. Return `-1` if this is not possible. This function does not raise exceptions.

double **Py_UNICODE_TONUMERIC**(*Py_UCS4* ch)

Return the character *ch* converted to a double. Return `-1.0` if this is not possible. This function does not raise exceptions.

These APIs can be used to work with surrogates:

int **Py_UNICODE_IS_SURROGATE**(*Py_UCS4* ch)

Check if *ch* is a surrogate (`0xD800 <= ch <= 0xDFFF`).

int **Py_UNICODE_IS_HIGH_SURROGATE**(*Py_UCS4* ch)

Check if *ch* is a high surrogate (`0xD800 <= ch <= 0xDBFF`).

int **Py_UNICODE_IS_LOW_SURROGATE**(*Py_UCS4* ch)

Check if *ch* is a low surrogate (`0xDC00 <= ch <= 0xDFFF`).

Py_UCS4 **Py_UNICODE_JOIN_SURROGATES**(*Py_UCS4* high, *Py_UCS4* low)

Join two surrogate code points and return a single `Py_UCS4` value. *high* and *low* are respectively the leading and trailing surrogates in a surrogate pair. *high* must be in the range [0xD800; 0xDBFF] and *low* must be in the range [0xDC00; 0xDFFF].

Creating and accessing Unicode strings

To create Unicode objects and access their basic sequence properties, use these APIs:

PyObject ***PyUnicode_New**(*Py_ssize_t* size, *Py_UCS4* maxchar)

Wartość zwracana: nowa referencja. Create a new Unicode object. *maxchar* should be the true maximum code point to be placed in the string. As an approximation, it can be rounded up to the nearest value in the sequence 127, 255, 65535, 1114111.

This is the recommended way to allocate a new Unicode object. Objects created using this function are not resizable.

On error, set an exception and return `NULL`.

Added in version 3.3.

PyObject ***PyUnicode_FromKindAndData**(int kind, const void *buffer, *Py_ssize_t* size)

Wartość zwracana: nowa referencja. Create a new Unicode object with the given *kind* (possible values are `PyUnicode_1BYTE_KIND` etc., as returned by `PyUnicode_KIND()`). The *buffer* must point to an array of *size* units of 1, 2 or 4 bytes per character, as given by the kind.

If necessary, the input *buffer* is copied and transformed into the canonical representation. For example, if the *buffer* is a UCS4 string (`PyUnicode_4BYTE_KIND`) and it consists only of codepoints in the UCS1 range, it will be transformed into UCS1 (`PyUnicode_1BYTE_KIND`).

Added in version 3.3.

PyObject ***PyUnicode_FromStringAndSize**(const char *str, *Py_ssize_t* size)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Create a Unicode object from the char buffer *str*. The bytes will be interpreted as being UTF-8 encoded. The buffer is copied into the new object. The return value might be a shared object, i.e. modification of the data is not allowed.

This function raises `SystemError` when:

- *size* < 0,
- *str* is `NULL` and *size* > 0

Zmienione w wersji 3.12: *str* == `NULL` with *size* > 0 is not allowed anymore.

PyObject ***PyUnicode_FromString**(const char *str)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Create a Unicode object from a UTF-8 encoded null-terminated char buffer *str*.

PyObject ***PyUnicode_FromFormat**(const char *format, ...)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Take a C `printf()`-style *format* string and a variable number of arguments, calculate the size of the resulting Python Unicode string and return a string with the values formatted into it. The variable arguments must be C types and must correspond exactly to the format characters in the *format* ASCII-encoded string.

A conversion specifier contains two or more characters and has the following components, which must occur in this order:

1. The `'%'` character, which marks the start of the specifier.
2. Conversion flags (optional), which affect the result of some conversion types.
3. Minimum field width (optional). If specified as an `'*'` (asterisk), the actual width is given in the next argument, which must be of type `int`, and the object to convert comes after the minimum field width and optional precision.
4. Precision (optional), given as a `'.'` (dot) followed by the precision. If specified as `'*'` (an asterisk), the actual precision is given in the next argument, which must be of type `int`, and the value to convert comes after the precision.
5. Length modifier (optional).
6. Conversion type.

The conversion flag characters are:

Flag	Meaning
`0`	The conversion will be zero padded for numeric values.
`-`	The converted value is left adjusted (overrides the `0` flag if both are given).

The length modifiers for following integer conversions (`d`, `i`, `o`, `u`, `x`, or `X`) specify the type of the argument (`int` by default):

Modifier	Types
`l`	`long` or `unsigned long`
`ll`	`long long` or `unsigned long long`
`j`	`intmax_t` or `uintmax_t`
`z`	`size_t` or `ssize_t`
`t`	`ptrdiff_t`

The length modifier `l` for following conversions `s` or `V` specify that the type of the argument is `const wchar_t*`.

The conversion specifiers are:

Conversion Specifier	Type	Comment
`%`	*n/a*	The literal `%` character.
`d`, `i`	Specified by the length modifier	The decimal representation of a signed C integer.
`u`	Specified by the length modifier	The decimal representation of an unsigned C integer.
`o`	Specified by the length modifier	The octal representation of an unsigned C integer.
`x`	Specified by the length modifier	The hexadecimal representation of an unsigned C integer (lowercase).
`X`	Specified by the length modifier	The hexadecimal representation of an unsigned C integer (uppercase).
`c`	`int`	A single character.
`s`	`const char*` or `const wchar_t*`	A null-terminated C character array.
`p`	`const void*`	The hex representation of a C pointer. Mostly equivalent to `printf("%p")` except that it is guaranteed to start with the literal `0x` regardless of what the platform's `printf` yields.
`A`	`PyObject*`	The result of calling `ascii()`.
`U`	`PyObject*`	A Unicode object.
`V`	`PyObject*`, `const char*` or `const wchar_t*`	A Unicode object (which may be `NULL`) and a null-terminated C character array as a second parameter (which will be used, if the first parameter is `NULL`).
`S`	`PyObject*`	The result of calling `PyObject_Str()`.
`R`	`PyObject*`	The result of calling `PyObject_Repr()`.
`T`	`PyObject*`	Get the fully qualified name of an object type; call `PyType_GetFullyQualifiedName()`.
`#T`	`PyObject*`	Similar to `T` format, but use a colon (`:`) as separator between the module name and the qualified name.
`N`	`PyTypeObject*`	Get the fully qualified name of a type; call `PyType_GetFullyQualifiedName()`.
`#N`	`PyTypeObject*`	Similar to `N` format, but use a colon (`:`) as separator between the module name and the qualified name.

Informacja

The width formatter unit is number of characters rather than bytes. The precision formatter unit is number of bytes or `wchar_t` items (if the length modifier `l` is used) for `"%s"` and `"%V"` (if the `PyObject*` argument is `NULL`), and a number of characters for `"%A"`, `"%U"`, `"%S"`, `"%R"` and `"%V"` (if the `PyObject*` argument is not `NULL`).

Informacja

Unlike to C `printf()` the `0` flag has effect even when a precision is given for integer conversions (`d`, `i`, `u`, `o`, `x`, or `X`).

Zmienione w wersji 3.2: Support for `"%lld"` and `"%llu"` added.

Zmienione w wersji 3.3: Support for `"%li"`, `"%lli"` and `"%zi"` added.

Zmienione w wersji 3.4: Support width and precision formatter for `"%s"`, `"%A"`, `"%U"`, `"%V"`, `"%S"`, `"%R"`

added.

Zmienione w wersji 3.12: Support for conversion specifiers `o` and `X`. Support for length modifiers `j` and `t`. Length modifiers are now applied to all integer conversions. Length modifier `l` is now applied to conversion specifiers `s` and `V`. Support for variable width and precision `*`. Support for flag `-`.

An unrecognized format character now sets a `SystemError`. In previous versions it caused all the rest of the format string to be copied as-is to the result string, and any extra arguments discarded.

Zmienione w wersji 3.13: Support for `%T`, `%#T`, `%N` and `%#N` formats added.

PyObject ***PyUnicode_FromFormatV**(const char *format, va_list vargs)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Identical to `PyUnicode_FromFormat()` except that it takes exactly two arguments.

PyObject ***PyUnicode_FromObject**(*PyObject* *obj)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Copy an instance of a Unicode subtype to a new true Unicode object if necessary. If *obj* is already a true Unicode object (not a subtype), return a new *strong reference* to the object.

Objects other than Unicode or its subtypes will cause a `TypeError`.

PyObject ***PyUnicode_FromEncodedObject**(*PyObject* *obj, const char *encoding, const char *errors)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Decode an encoded object *obj* to a Unicode object.

`bytes`, `bytearray` and other *bytes-like objects* are decoded according to the given *encoding* and using the error handling defined by *errors*. Both can be `NULL` to have the interface use the default values (see *Built-in Codecs* for details).

All other objects, including Unicode objects, cause a `TypeError` to be set.

The API returns `NULL` if there was an error. The caller is responsible for decref'ing the returned objects.

Py_ssize_t **PyUnicode_GetLength**(*PyObject* *unicode)

Część stabilnego ABI *od wersji 3.7.* Return the length of the Unicode object, in code points.

On error, set an exception and return `-1`.

Added in version 3.3.

Py_ssize_t **PyUnicode_CopyCharacters**(*PyObject* *to, *Py_ssize_t* to_start, *PyObject* *from, *Py_ssize_t* from_start, *Py_ssize_t* how_many)

Copy characters from one Unicode object into another. This function performs character conversion when necessary and falls back to `memcpy()` if possible. Returns `-1` and sets an exception on error, otherwise returns the number of copied characters.

Added in version 3.3.

Py_ssize_t **PyUnicode_Fill**(*PyObject* *unicode, *Py_ssize_t* start, *Py_ssize_t* length, *Py_UCS4* fill_char)

Fill a string with a character: write *fill_char* into `unicode[start:start+length]`.

Fail if *fill_char* is bigger than the string maximum character, or if the string has more than 1 reference.

Return the number of written character, or return `-1` and raise an exception on error.

Added in version 3.3.

int **PyUnicode_WriteChar**(*PyObject* *unicode, *Py_ssize_t* index, *Py_UCS4* character)

Część stabilnego ABI *od wersji 3.7.* Write a character to a string. The string must have been created through `PyUnicode_New()`. Since Unicode strings are supposed to be immutable, the string must not be shared, or have been hashed yet.

This function checks that *unicode* is a Unicode object, that the index is not out of bounds, and that the object can be modified safely (i.e. that it its reference count is one).

Return `0` on success, `-1` on error with an exception set.

Added in version 3.3.

Py_UCS4 **PyUnicode_ReadChar**(*PyObject* *unicode, *Py_ssize_t* index)

Część stabilnego ABI *od wersji 3.7.* Read a character from a string. This function checks that *unicode* is a Unicode object and the index is not out of bounds, in contrast to `PyUnicode_READ_CHAR()`, which performs no error checking.

Return character on success, `-1` on error with an exception set.

Added in version 3.3.

PyObject ***PyUnicode_Substring**(*PyObject* *unicode, *Py_ssize_t* start, *Py_ssize_t* end)

Wartość zwracana: nowa referencja. Część stabilnego ABI *od wersji 3.7.* Return a substring of *unicode*, from character index *start* (included) to character index *end* (excluded). Negative indices are not supported. On error, set an exception and return `NULL`.

Added in version 3.3.

Py_UCS4 ***PyUnicode_AsUCS4**(*PyObject* *unicode, *Py_UCS4* *buffer, *Py_ssize_t* buflen, int copy_null)

Część stabilnego ABI *od wersji 3.7.* Copy the string *unicode* into a UCS4 buffer, including a null character, if *copy_null* is set. Returns `NULL` and sets an exception on error (in particular, a `SystemError` if *buflen* is smaller than the length of *unicode*). *buffer* is returned on success.

Added in version 3.3.

Py_UCS4 ***PyUnicode_AsUCS4Copy**(*PyObject* *unicode)

Część stabilnego ABI *od wersji 3.7.* Copy the string *unicode* into a new UCS4 buffer that is allocated using `PyMem_Malloc()`. If this fails, `NULL` is returned with a `MemoryError` set. The returned buffer always has an extra null code point appended.

Added in version 3.3.

Locale Encoding

The current locale encoding can be used to decode text from the operating system.

PyObject ***PyUnicode_DecodeLocaleAndSize**(const char *str, *Py_ssize_t* length, const char *errors)

Wartość zwracana: nowa referencja. Część stabilnego ABI *od wersji 3.7.* Decode a string from UTF-8 on Android and VxWorks, or from the current locale encoding on other platforms. The supported error handlers are `"strict"` and `"surrogateescape"` (**PEP 383**). The decoder uses `"strict"` error handler if *errors* is `NULL`. *str* must end with a null character but cannot contain embedded null characters.

Use `PyUnicode_DecodeFSDefaultAndSize()` to decode a string from the *filesystem encoding and error handler*.

This function ignores the Python UTF-8 Mode.

> **Zobacz także**
>
> The `Py_DecodeLocale()` function.

Added in version 3.3.

Zmienione w wersji 3.7: The function now also uses the current locale encoding for the `surrogateescape` error handler, except on Android. Previously, `Py_DecodeLocale()` was used for the `surrogateescape`, and the current locale encoding was used for `strict`.

PyObject ***PyUnicode_DecodeLocale**(const char *str, const char *errors)

Wartość zwracana: nowa referencja. Część stabilnego ABI *od wersji 3.7.* Similar to `PyUnicode_DecodeLocaleAndSize()`, but compute the string length using `strlen()`.

Added in version 3.3.

PyObject ***PyUnicode_EncodeLocale*** (*PyObject* *unicode, const char *errors)

Wartość zwracana: nowa referencja. Część stabilnego ABI *od wersji 3.7.* Encode a Unicode object to UTF-8 on Android and VxWorks, or to the current locale encoding on other platforms. The supported error handlers are `"strict"` and `"surrogateescape"` (**PEP 383**). The encoder uses `"strict"` error handler if *errors* is `NULL`. Return a `bytes` object. *unicode* cannot contain embedded null characters.

Use `PyUnicode_EncodeFSDefault()` to encode a string to the *filesystem encoding and error handler*.

This function ignores the Python UTF-8 Mode.

> **Zobacz także**
>
> The `Py_EncodeLocale()` function.

Added in version 3.3.

Zmienione w wersji 3.7: The function now also uses the current locale encoding for the `surrogateescape` error handler, except on Android. Previously, `Py_EncodeLocale()` was used for the `surrogateescape`, and the current locale encoding was used for `strict`.

File System Encoding

Functions encoding to and decoding from the *filesystem encoding and error handler* (**PEP 383** and **PEP 529**).

To encode file names to `bytes` during argument parsing, the `"O&"` converter should be used, passing `PyUnicode_FSConverter()` as the conversion function:

int ***PyUnicode_FSConverter*** (*PyObject* *obj, void *result)

Część stabilnego ABI. ParseTuple converter: encode `str` objects – obtained directly or through the `os.PathLike` interface – to `bytes` using `PyUnicode_EncodeFSDefault()`; `bytes` objects are output as-is. *result* must be a `PyBytesObject*` which must be released when it is no longer used.

Added in version 3.1.

Zmienione w wersji 3.6: Accepts a *path-like object*.

To decode file names to `str` during argument parsing, the `"O&"` converter should be used, passing `PyUnicode_FSDecoder()` as the conversion function:

int ***PyUnicode_FSDecoder*** (*PyObject* *obj, void *result)

Część stabilnego ABI. ParseTuple converter: decode `bytes` objects – obtained either directly or indirectly through the `os.PathLike` interface – to `str` using `PyUnicode_DecodeFSDefaultAndSize()`; `str` objects are output as-is. *result* must be a `PyUnicodeObject*` which must be released when it is no longer used.

Added in version 3.2.

Zmienione w wersji 3.6: Accepts a *path-like object*.

PyObject ***PyUnicode_DecodeFSDefaultAndSize*** (const char *str, *Py_ssize_t* size)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Decode a string from the *filesystem encoding and error handler*.

If you need to decode a string from the current locale encoding, use `PyUnicode_DecodeLocaleAndSize()`.

> **Zobacz także**
>
> The `Py_DecodeLocale()` function.

Zmienione w wersji 3.6: The *filesystem error handler* is now used.

PyObject ***PyUnicode_DecodeFSDefault*** (const char *str)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Decode a null-terminated string from the *filesystem encoding and error handler*.

If the string length is known, use `PyUnicode_DecodeFSDefaultAndSize()`.

Zmienione w wersji 3.6: The *filesystem error handler* is now used.

PyObject ***PyUnicode_EncodeFSDefault*** (*PyObject* *unicode)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Encode a Unicode object to the *filesystem encoding and error handler*, and return `bytes`. Note that the resulting `bytes` object can contain null bytes.

If you need to encode a string to the current locale encoding, use `PyUnicode_EncodeLocale()`.

Zobacz także

The `Py_EncodeLocale()` function.

Added in version 3.2.

Zmienione w wersji 3.6: The *filesystem error handler* is now used.

wchar_t Support

`wchar_t` support for platforms which support it:

PyObject ***PyUnicode_FromWideChar*** (const wchar_t *wstr, *Py_ssize_t* size)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Create a Unicode object from the `wchar_t` buffer *wstr* of the given *size*. Passing `-1` as the *size* indicates that the function must itself compute the length, using `wcslen()`. Return `NULL` on failure.

Py_ssize_t ***PyUnicode_AsWideChar*** (*PyObject* *unicode, wchar_t *wstr, *Py_ssize_t* size)

Część stabilnego ABI. Copy the Unicode object contents into the `wchar_t` buffer *wstr*. At most *size* `wchar_t` characters are copied (excluding a possibly trailing null termination character). Return the number of `wchar_t` characters copied or `-1` in case of an error.

When *wstr* is `NULL`, instead return the *size* that would be required to store all of *unicode* including a terminating null.

Note that the resulting `wchar_t*` string may or may not be null-terminated. It is the responsibility of the caller to make sure that the `wchar_t*` string is null-terminated in case this is required by the application. Also, note that the `wchar_t*` string might contain null characters, which would cause the string to be truncated when used with most C functions.

wchar_t ***PyUnicode_AsWideCharString*** (*PyObject* *unicode, *Py_ssize_t* *size)

Część stabilnego ABI *od wersji 3.7.* Convert the Unicode object to a wide character string. The output string always ends with a null character. If *size* is not `NULL`, write the number of wide characters (excluding the trailing null termination character) into **size*. Note that the resulting `wchar_t` string might contain null characters, which would cause the string to be truncated when used with most C functions. If *size* is `NULL` and the `wchar_t*` string contains null characters a `ValueError` is raised.

Returns a buffer allocated by `PyMem_New` (use `PyMem_Free()` to free it) on success. On error, returns `NULL` and **size* is undefined. Raises a `MemoryError` if memory allocation is failed.

Added in version 3.2.

Zmienione w wersji 3.7: Raises a `ValueError` if *size* is `NULL` and the `wchar_t*` string contains null characters.

Built-in Codecs

Python provides a set of built-in codecs which are written in C for speed. All of these codecs are directly usable via the following functions.

Many of the following APIs take two arguments encoding and errors, and they have the same semantics as the ones of the built-in `str()` string object constructor.

Setting encoding to `NULL` causes the default encoding to be used which is UTF-8. The file system calls should use `PyUnicode_FSConverter()` for encoding file names. This uses the *filesystem encoding and error handler* internally.

Error handling is set by errors which may also be set to `NULL` meaning to use the default handling defined for the codec. Default error handling for all built-in codecs is „strict" (`ValueError` is raised).

The codecs all use a similar interface. Only deviations from the following generic ones are documented for simplicity.

Generic Codecs

These are the generic codec APIs:

PyObject ***PyUnicode_Decode**(const char *str, *Py_ssize_t* size, const char *encoding, const char *errors)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Create a Unicode object by decoding *size* bytes of the encoded string *str*. *encoding* and *errors* have the same meaning as the parameters of the same name in the `str()` built-in function. The codec to be used is looked up using the Python codec registry. Return `NULL` if an exception was raised by the codec.

PyObject ***PyUnicode_AsEncodedString**(*PyObject* *unicode, const char *encoding, const char *errors)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Encode a Unicode object and return the result as Python bytes object. *encoding* and *errors* have the same meaning as the parameters of the same name in the Unicode `encode()` method. The codec to be used is looked up using the Python codec registry. Return `NULL` if an exception was raised by the codec.

UTF-8 Codecs

These are the UTF-8 codec APIs:

PyObject ***PyUnicode_DecodeUTF8**(const char *str, *Py_ssize_t* size, const char *errors)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Create a Unicode object by decoding *size* bytes of the UTF-8 encoded string *str*. Return `NULL` if an exception was raised by the codec.

PyObject ***PyUnicode_DecodeUTF8Stateful**(const char *str, *Py_ssize_t* size, const char *errors, *Py_ssize_t* *consumed)

Wartość zwracana: nowa referencja. Część stabilnego ABI. If *consumed* is `NULL`, behave like `PyUnicode_DecodeUTF8()`. If *consumed* is not `NULL`, trailing incomplete UTF-8 byte sequences will not be treated as an error. Those bytes will not be decoded and the number of bytes that have been decoded will be stored in *consumed*.

PyObject ***PyUnicode_AsUTF8String**(*PyObject* *unicode)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Encode a Unicode object using UTF-8 and return the result as Python bytes object. Error handling is „strict". Return `NULL` if an exception was raised by the codec.

The function fails if the string contains surrogate code points (`U+D800` - `U+DFFF`).

const char ***PyUnicode_AsUTF8AndSize**(*PyObject* *unicode, *Py_ssize_t* *size)

Część stabilnego ABI *od wersji 3.10.* Return a pointer to the UTF-8 encoding of the Unicode object, and store the size of the encoded representation (in bytes) in *size*. The *size* argument can be `NULL`; in this case no size will be stored. The returned buffer always has an extra null byte appended (not included in *size*), regardless of whether there are any other null code points.

On error, set an exception, set *size* to `-1` (if it's not NULL) and return `NULL`.

The function fails if the string contains surrogate code points (`U+D800` - `U+DFFF`).

This caches the UTF-8 representation of the string in the Unicode object, and subsequent calls will return a pointer to the same buffer. The caller is not responsible for deallocating the buffer. The buffer is deallocated and pointers to it become invalid when the Unicode object is garbage collected.

Added in version 3.3.

Zmienione w wersji 3.7: The return type is now `const char *` rather of `char *`.

Zmienione w wersji 3.10: This function is a part of the *limited API*.

const char ***PyUnicode_AsUTF8**(*PyObject* *unicode)

As `PyUnicode_AsUTF8AndSize()`, but does not store the size.

Added in version 3.3.

Zmienione w wersji 3.7: The return type is now `const char *` rather of `char *`.

UTF-32 Codecs

These are the UTF-32 codec APIs:

PyObject ***PyUnicode_DecodeUTF32**(const char *str, *Py_ssize_t* size, const char *errors, int *byteorder)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Decode *size* bytes from a UTF-32 encoded buffer string and return the corresponding Unicode object. *errors* (if non-`NULL`) defines the error handling. It defaults to „strict".

If *byteorder* is non-`NULL`, the decoder starts decoding using the given byte order:

```
*byteorder == -1: little endian
*byteorder == 0:  native order
*byteorder == 1:  big endian
```

If `*byteorder` is zero, and the first four bytes of the input data are a byte order mark (BOM), the decoder switches to this byte order and the BOM is not copied into the resulting Unicode string. If `*byteorder` is `-1` or `1`, any byte order mark is copied to the output.

After completion, **byteorder* is set to the current byte order at the end of input data.

If *byteorder* is `NULL`, the codec starts in native order mode.

Return `NULL` if an exception was raised by the codec.

PyObject ***PyUnicode_DecodeUTF32Stateful**(const char *str, *Py_ssize_t* size, const char *errors, int *byteorder, *Py_ssize_t* *consumed)

Wartość zwracana: nowa referencja. Część stabilnego ABI. If *consumed* is `NULL`, behave like `PyUnicode_DecodeUTF32()`. If *consumed* is not `NULL`, `PyUnicode_DecodeUTF32Stateful()` will not treat trailing incomplete UTF-32 byte sequences (such as a number of bytes not divisible by four) as an error. Those bytes will not be decoded and the number of bytes that have been decoded will be stored in *consumed*.

PyObject ***PyUnicode_AsUTF32String**(*PyObject* *unicode)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return a Python byte string using the UTF-32 encoding in native byte order. The string always starts with a BOM mark. Error handling is „strict". Return `NULL` if an exception was raised by the codec.

UTF-16 Codecs

These are the UTF-16 codec APIs:

PyObject ***PyUnicode_DecodeUTF16**(const char *str, *Py_ssize_t* size, const char *errors, int *byteorder)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Decode *size* bytes from a UTF-16 encoded buffer string and return the corresponding Unicode object. *errors* (if non-`NULL`) defines the error handling. It defaults to „strict".

If *byteorder* is non-`NULL`, the decoder starts decoding using the given byte order:

```
*byteorder == -1: little endian
*byteorder == 0:  native order
*byteorder == 1:  big endian
```

If `*byteorder` is zero, and the first two bytes of the input data are a byte order mark (BOM), the decoder switches to this byte order and the BOM is not copied into the resulting Unicode string. If `*byteorder` is `-1` or `1`, any byte order mark is copied to the output (where it will result in either a `\ufeff` or a `\ufffe` character).

After completion, `*byteorder` is set to the current byte order at the end of input data.

If *byteorder* is `NULL`, the codec starts in native order mode.

Return `NULL` if an exception was raised by the codec.

PyObject ***PyUnicode_DecodeUTF16Stateful***(const char *str, *Py_ssize_t* size, const char *errors, int *byteorder, *Py_ssize_t* *consumed)

Wartość zwracana: nowa referencja. Część stabilnego ABI. If *consumed* is `NULL`, behave like `PyUnicode_DecodeUTF16()`. If *consumed* is not `NULL`, `PyUnicode_DecodeUTF16Stateful()` will not treat trailing incomplete UTF-16 byte sequences (such as an odd number of bytes or a split surrogate pair) as an error. Those bytes will not be decoded and the number of bytes that have been decoded will be stored in *consumed*.

PyObject ***PyUnicode_AsUTF16String***(*PyObject* *unicode)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return a Python byte string using the UTF-16 encoding in native byte order. The string always starts with a BOM mark. Error handling is „strict”. Return `NULL` if an exception was raised by the codec.

UTF-7 Codecs

These are the UTF-7 codec APIs:

PyObject ***PyUnicode_DecodeUTF7***(const char *str, *Py_ssize_t* size, const char *errors)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Create a Unicode object by decoding *size* bytes of the UTF-7 encoded string *str*. Return `NULL` if an exception was raised by the codec.

PyObject ***PyUnicode_DecodeUTF7Stateful***(const char *str, *Py_ssize_t* size, const char *errors, *Py_ssize_t* *consumed)

Wartość zwracana: nowa referencja. Część stabilnego ABI. If *consumed* is `NULL`, behave like `PyUnicode_DecodeUTF7()`. If *consumed* is not `NULL`, trailing incomplete UTF-7 base-64 sections will not be treated as an error. Those bytes will not be decoded and the number of bytes that have been decoded will be stored in *consumed*.

Unicode-Escape Codecs

These are the „Unicode Escape” codec APIs:

PyObject ***PyUnicode_DecodeUnicodeEscape***(const char *str, *Py_ssize_t* size, const char *errors)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Create a Unicode object by decoding *size* bytes of the Unicode-Escape encoded string *str*. Return `NULL` if an exception was raised by the codec.

PyObject ***PyUnicode_AsUnicodeEscapeString***(*PyObject* *unicode)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Encode a Unicode object using Unicode-Escape and return the result as a bytes object. Error handling is „strict”. Return `NULL` if an exception was raised by the codec.

Raw-Unicode-Escape Codecs

These are the „Raw Unicode Escape” codec APIs:

PyObject ***PyUnicode_DecodeRawUnicodeEscape**(const char *str, *Py_ssize_t* size, const char *errors)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Create a Unicode object by decoding *size* bytes of the Raw-Unicode-Escape encoded string *str*. Return `NULL` if an exception was raised by the codec.

PyObject ***PyUnicode_AsRawUnicodeEscapeString**(*PyObject* *unicode)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Encode a Unicode object using Raw-Unicode--Escape and return the result as a bytes object. Error handling is „strict”. Return `NULL` if an exception was raised by the codec.

Latin-1 Codecs

These are the Latin-1 codec APIs: Latin-1 corresponds to the first 256 Unicode ordinals and only these are accepted by the codecs during encoding.

PyObject ***PyUnicode_DecodeLatin1**(const char *str, *Py_ssize_t* size, const char *errors)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Create a Unicode object by decoding *size* bytes of the Latin-1 encoded string *str*. Return `NULL` if an exception was raised by the codec.

PyObject ***PyUnicode_AsLatin1String**(*PyObject* *unicode)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Encode a Unicode object using Latin-1 and return the result as Python bytes object. Error handling is „strict”. Return `NULL` if an exception was raised by the codec.

ASCII Codecs

These are the ASCII codec APIs. Only 7-bit ASCII data is accepted. All other codes generate errors.

PyObject ***PyUnicode_DecodeASCII**(const char *str, *Py_ssize_t* size, const char *errors)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Create a Unicode object by decoding *size* bytes of the ASCII encoded string *str*. Return `NULL` if an exception was raised by the codec.

PyObject ***PyUnicode_AsASCIIString**(*PyObject* *unicode)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Encode a Unicode object using ASCII and return the result as Python bytes object. Error handling is „strict”. Return `NULL` if an exception was raised by the codec.

Character Map Codecs

This codec is special in that it can be used to implement many different codecs (and this is in fact what was done to obtain most of the standard codecs included in the `encodings` package). The codec uses mappings to encode and decode characters. The mapping objects provided must support the `__getitem__()` mapping interface; dictionaries and sequences work well.

These are the mapping codec APIs:

PyObject ***PyUnicode_DecodeCharmap**(const char *str, *Py_ssize_t* length, *PyObject* *mapping, const char *errors)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Create a Unicode object by decoding *size* bytes of the encoded string *str* using the given *mapping* object. Return `NULL` if an exception was raised by the codec.

If *mapping* is `NULL`, Latin-1 decoding will be applied. Else *mapping* must map bytes ordinals (integers in the range from 0 to 255) to Unicode strings, integers (which are then interpreted as Unicode ordinals) or `None`. Unmapped data bytes – ones which cause a `LookupError`, as well as ones which get mapped to `None`, `0xFFFE` or `'\ufffe'`, are treated as undefined mappings and cause an error.

PyObject ***PyUnicode_AsCharmapString*** (*PyObject* *unicode, *PyObject* *mapping)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Encode a Unicode object using the given *mapping* object and return the result as a bytes object. Error handling is „strict". Return `NULL` if an exception was raised by the codec.

The *mapping* object must map Unicode ordinal integers to bytes objects, integers in the range from 0 to 255 or `None`. Unmapped character ordinals (ones which cause a `LookupError`) as well as mapped to `None` are treated as „undefined mapping" and cause an error.

The following codec API is special in that maps Unicode to Unicode.

PyObject ***PyUnicode_Translate*** (*PyObject* *unicode, *PyObject* *table, const char *errors)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Translate a string by applying a character mapping table to it and return the resulting Unicode object. Return `NULL` if an exception was raised by the codec.

The mapping table must map Unicode ordinal integers to Unicode ordinal integers or `None` (causing deletion of the character).

Mapping tables need only provide the `__getitem__()` interface; dictionaries and sequences work well. Unmapped character ordinals (ones which cause a `LookupError`) are left untouched and are copied as-is.

errors has the usual meaning for codecs. It may be `NULL` which indicates to use the default error handling.

MBCS codecs for Windows

These are the MBCS codec APIs. They are currently only available on Windows and use the Win32 MBCS converters to implement the conversions. Note that MBCS (or DBCS) is a class of encodings, not just one. The target encoding is defined by the user settings on the machine running the codec.

PyObject ***PyUnicode_DecodeMBCS*** (const char *str, *Py_ssize_t* size, const char *errors)

Wartość zwracana: nowa referencja. Część stabilnego ABI *on Windows od wersji 3.7.* Create a Unicode object by decoding *size* bytes of the MBCS encoded string *str*. Return `NULL` if an exception was raised by the codec.

PyObject ***PyUnicode_DecodeMBCSStateful*** (const char *str, *Py_ssize_t* size, const char *errors, *Py_ssize_t* *consumed)

Wartość zwracana: nowa referencja. Część stabilnego ABI *on Windows od wersji 3.7.* If *consumed* is `NULL`, behave like `PyUnicode_DecodeMBCS()`. If *consumed* is not `NULL`, `PyUnicode_DecodeMBCSStateful()` will not decode trailing lead byte and the number of bytes that have been decoded will be stored in *consumed*.

PyObject ***PyUnicode_AsMBCSString*** (*PyObject* *unicode)

Wartość zwracana: nowa referencja. Część stabilnego ABI *on Windows od wersji 3.7.* Encode a Unicode object using MBCS and return the result as Python bytes object. Error handling is „strict". Return `NULL` if an exception was raised by the codec.

PyObject ***PyUnicode_EncodeCodePage*** (int code_page, *PyObject* *unicode, const char *errors)

Wartość zwracana: nowa referencja. Część stabilnego ABI *on Windows od wersji 3.7.* Encode the Unicode object using the specified code page and return a Python bytes object. Return `NULL` if an exception was raised by the codec. Use `CP_ACP` code page to get the MBCS encoder.

Added in version 3.3.

Methods & Slots

Methods and Slot Functions

The following APIs are capable of handling Unicode objects and strings on input (we refer to them as strings in the descriptions) and return Unicode objects or integers as appropriate.

They all return `NULL` or `-1` if an exception occurs.

PyObject ***PyUnicode_Concat*** (*PyObject* *left, *PyObject* *right)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Concat two strings giving a new Unicode string.

PyObject ***PyUnicode_Split*** (*PyObject* *unicode, *PyObject* *sep, *Py_ssize_t* maxsplit)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Split a string giving a list of Unicode strings. If *sep* is `NULL`, splitting will be done at all whitespace substrings. Otherwise, splits occur at the given separator. At most *maxsplit* splits will be done. If negative, no limit is set. Separators are not included in the resulting list.

PyObject ***PyUnicode_Splitlines*** (*PyObject* *unicode, int keepends)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Split a Unicode string at line breaks, returning a list of Unicode strings. CRLF is considered to be one line break. If *keepends* is `0`, the Line break characters are not included in the resulting strings.

PyObject ***PyUnicode_Join*** (*PyObject* *separator, *PyObject* *seq)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Join a sequence of strings using the given *separator* and return the resulting Unicode string.

Py_ssize_t ***PyUnicode_Tailmatch*** (*PyObject* *unicode, *PyObject* *substr, *Py_ssize_t* start, *Py_ssize_t* end, int direction)

Część stabilnego ABI. Return `1` if *substr* matches `unicode[start:end]` at the given tail end (*direction* == `-1` means to do a prefix match, *direction* == `1` a suffix match), `0` otherwise. Return `-1` if an error occurred.

Py_ssize_t ***PyUnicode_Find*** (*PyObject* *unicode, *PyObject* *substr, *Py_ssize_t* start, *Py_ssize_t* end, int direction)

Część stabilnego ABI. Return the first position of *substr* in `unicode[start:end]` using the given *direction* (*direction* == `1` means to do a forward search, *direction* == `-1` a backward search). The return value is the index of the first match; a value of `-1` indicates that no match was found, and `-2` indicates that an error occurred and an exception has been set.

Py_ssize_t ***PyUnicode_FindChar*** (*PyObject* *unicode, *Py_UCS4* ch, *Py_ssize_t* start, *Py_ssize_t* end, int direction)

Część stabilnego ABI *od wersji 3.7.* Return the first position of the character *ch* in `unicode[start:end]` using the given *direction* (*direction* == `1` means to do a forward search, *direction* == `-1` a backward search). The return value is the index of the first match; a value of `-1` indicates that no match was found, and `-2` indicates that an error occurred and an exception has been set.

Added in version 3.3.

Zmienione w wersji 3.7: *start* and *end* are now adjusted to behave like `unicode[start:end]`.

Py_ssize_t ***PyUnicode_Count*** (*PyObject* *unicode, *PyObject* *substr, *Py_ssize_t* start, *Py_ssize_t* end)

Część stabilnego ABI. Return the number of non-overlapping occurrences of *substr* in `unicode[start:end]`. Return `-1` if an error occurred.

PyObject ***PyUnicode_Replace*** (*PyObject* *unicode, *PyObject* *substr, *PyObject* *replstr, *Py_ssize_t* maxcount)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Replace at most *maxcount* occurrences of *substr* in *unicode* with *replstr* and return the resulting Unicode object. *maxcount* == `-1` means replace all occurrences.

int ***PyUnicode_Compare*** (*PyObject* *left, *PyObject* *right)

Część stabilnego ABI. Compare two strings and return `-1`, `0`, `1` for less than, equal, and greater than, respectively.

This function returns `-1` upon failure, so one should call `PyErr_Occurred()` to check for errors.

int ***PyUnicode_EqualToUTF8AndSize*** (*PyObject* *unicode, const char *string, *Py_ssize_t* size)

Część stabilnego ABI *od wersji 3.13.* Compare a Unicode object with a char buffer which is interpreted as being UTF-8 or ASCII encoded and return true (`1`) if they are equal, or false (`0`) otherwise. If the Unicode object contains surrogate code points (`U+D800` - `U+DFFF`) or the C string is not valid UTF-8, false (`0`) is returned.

This function does not raise exceptions.

Added in version 3.13.

int ***PyUnicode_EqualToUTF8*** (*PyObject* *unicode, const char *string)

Część stabilnego ABI *od wersji 3.13.* Similar to `PyUnicode_EqualToUTF8AndSize()`, but compute *string* length using `strlen()`. If the Unicode object contains null characters, false (`0`) is returned.

Added in version 3.13.

int **PyUnicode_CompareWithASCIIString**(*PyObject* *unicode, const char *string)

Część stabilnego ABI. Compare a Unicode object, *unicode*, with *string* and return `-1`, `0`, `1` for less than, equal, and greater than, respectively. It is best to pass only ASCII-encoded strings, but the function interprets the input string as ISO-8859-1 if it contains non-ASCII characters.

This function does not raise exceptions.

PyObject ***PyUnicode_RichCompare**(*PyObject* *left, *PyObject* *right, int op)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Rich compare two Unicode strings and return one of the following:

- `NULL` in case an exception was raised
- `Py_True` or `Py_False` for successful comparisons
- `Py_NotImplemented` in case the type combination is unknown

Possible values for *op* are `Py_GT`, `Py_GE`, `Py_EQ`, `Py_NE`, `Py_LT`, and `Py_LE`.

PyObject ***PyUnicode_Format**(*PyObject* *format, *PyObject* *args)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return a new string object from *format* and *args*; this is analogous to `format % args`.

int **PyUnicode_Contains**(*PyObject* *unicode, *PyObject* *substr)

Część stabilnego ABI. Check whether *substr* is contained in *unicode* and return true or false accordingly.

substr has to coerce to a one element Unicode string. `-1` is returned if there was an error.

void **PyUnicode_InternInPlace**(*PyObject* **p_unicode)

Część stabilnego ABI. Intern the argument `*p_unicode` in place. The argument must be the address of a pointer variable pointing to a Python Unicode string object. If there is an existing interned string that is the same as `*p_unicode`, it sets `*p_unicode` to it (releasing the reference to the old string object and creating a new *strong reference* to the interned string object), otherwise it leaves `*p_unicode` alone and interns it.

(Clarification: even though there is a lot of talk about references, think of this function as reference-neutral. You must own the object you pass in; after the call you no longer own the passed-in reference, but you newly own the result.)

This function never raises an exception. On error, it leaves its argument unchanged without interning it.

Instances of subclasses of `str` may not be interned, that is, `PyUnicode_CheckExact(*p_unicode)` must be true. If it is not, then – as with any other error – the argument is left unchanged.

Note that interned strings are not "immortal". You must keep a reference to the result to benefit from interning.

PyObject ***PyUnicode_InternFromString**(const char *str)

Wartość zwracana: nowa referencja. Część stabilnego ABI. A combination of `PyUnicode_FromString()` and `PyUnicode_InternInPlace()`, meant for statically allocated strings.

Return a new („owned") reference to either a new Unicode string object that has been interned, or an earlier interned string object with the same value.

Python may keep a reference to the result, or make it *immortal*, preventing it from being garbage-collected promptly. For interning an unbounded number of different strings, such as ones coming from user input, prefer calling `PyUnicode_FromString()` and `PyUnicode_InternInPlace()` directly.

Szczegół implementacyjny CPythona: Strings interned this way are made *immortal*.

8.3.4 Tuple Objects

type **PyTupleObject**

This subtype of `PyObject` represents a Python tuple object.

PyTypeObject **PyTuple_Type**

Część stabilnego ABI. This instance of `PyTypeObject` represents the Python tuple type; it is the same object as `tuple` in the Python layer.

int **PyTuple_Check**(*PyObject* *p)

Return true if *p* is a tuple object or an instance of a subtype of the tuple type. This function always succeeds.

int **PyTuple_CheckExact**(*PyObject* *p)

Return true if *p* is a tuple object, but not an instance of a subtype of the tuple type. This function always succeeds.

PyObject ***PyTuple_New**(*Py_ssize_t* len)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return a new tuple object of size *len*, or `NULL` with an exception set on failure.

PyObject ***PyTuple_Pack**(*Py_ssize_t* n, ...)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return a new tuple object of size *n*, or `NULL` with an exception set on failure. The tuple values are initialized to the subsequent *n* C arguments pointing to Python objects. `PyTuple_Pack(2, a, b)` is equivalent to `Py_BuildValue("(OO)", a, b)`.

Py_ssize_t **PyTuple_Size**(*PyObject* *p)

Część stabilnego ABI. Take a pointer to a tuple object, and return the size of that tuple. On error, return `-1` and with an exception set.

Py_ssize_t **PyTuple_GET_SIZE**(*PyObject* *p)

Like `PyTuple_Size()`, but without error checking.

PyObject ***PyTuple_GetItem**(*PyObject* *p, *Py_ssize_t* pos)

Wartość zwracana: pożyczona referencja. Część stabilnego ABI. Return the object at position *pos* in the tuple pointed to by *p*. If *pos* is negative or out of bounds, return `NULL` and set an `IndexError` exception.

The returned reference is borrowed from the tuple *p* (that is: it is only valid as long as you hold a reference to *p*). To get a *strong reference*, use `Py_NewRef(PyTuple_GetItem(...))` or `PySequence_GetItem()`.

PyObject ***PyTuple_GET_ITEM**(*PyObject* *p, *Py_ssize_t* pos)

Wartość zwracana: pożyczona referencja. Like `PyTuple_GetItem()`, but does no checking of its arguments.

PyObject ***PyTuple_GetSlice**(*PyObject* *p, *Py_ssize_t* low, *Py_ssize_t* high)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return the slice of the tuple pointed to by *p* between *low* and *high*, or `NULL` with an exception set on failure.

This is the equivalent of the Python expression `p[low:high]`. Indexing from the end of the tuple is not supported.

int **PyTuple_SetItem**(*PyObject* *p, *Py_ssize_t* pos, *PyObject* *o)

Część stabilnego ABI. Insert a reference to object *o* at position *pos* of the tuple pointed to by *p*. Return `0` on success. If *pos* is out of bounds, return `-1` and set an `IndexError` exception.

> **Informacja**
>
> This function „steals” a reference to *o* and discards a reference to an item already in the tuple at the affected position.

void **PyTuple_SET_ITEM**(*PyObject* *p, *Py_ssize_t* pos, *PyObject* *o)

Like `PyTuple_SetItem()`, but does no error checking, and should *only* be used to fill in brand new tuples.

Bounds checking is performed as an assertion if Python is built in debug mode or `with assertions`.

> **Informacja**
>
> This function „steals” a reference to *o*, and, unlike `PyTuple_SetItem()`, does *not* discard a reference to any item that is being replaced; any reference in the tuple at position *pos* will be leaked.

int **`_PyTuple_Resize`**(*PyObject* **p, *Py_ssize_t* newsize)

Can be used to resize a tuple. *newsize* will be the new length of the tuple. Because tuples are *supposed* to be immutable, this should only be used if there is only one reference to the object. Do *not* use this if the tuple may already be known to some other part of the code. The tuple will always grow or shrink at the end. Think of this as destroying the old tuple and creating a new one, only more efficiently. Returns `0` on success. Client code should never assume that the resulting value of `*p` will be the same as before calling this function. If the object referenced by `*p` is replaced, the original `*p` is destroyed. On failure, returns `-1` and sets `*p` to `NULL`, and raises `MemoryError` or `SystemError`.

8.3.5 Struct Sequence Objects

Struct sequence objects are the C equivalent of `namedtuple()` objects, i.e. a sequence whose items can also be accessed through attributes. To create a struct sequence, you first have to create a specific struct sequence type.

PyTypeObject ***PyStructSequence_NewType**(*PyStructSequence_Desc* *desc)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Create a new struct sequence type from the data in *desc*, described below. Instances of the resulting type can be created with `PyStructSequence_New()`.

Return `NULL` with an exception set on failure.

void **`PyStructSequence_InitType`**(*PyTypeObject* *type, *PyStructSequence_Desc* *desc)

Initializes a struct sequence type *type* from *desc* in place.

int **`PyStructSequence_InitType2`**(*PyTypeObject* *type, *PyStructSequence_Desc* *desc)

Like `PyStructSequence_InitType()`, but returns `0` on success and `-1` with an exception set on failure.

Added in version 3.4.

type **`PyStructSequence_Desc`**

Część stabilnego ABI *(w tym wszystkie składniki).* Contains the meta information of a struct sequence type to create.

const char ***name**

Fully qualified name of the type; null-terminated UTF-8 encoded. The name must contain the module name.

const char ***doc**

Pointer to docstring for the type or `NULL` to omit.

PyStructSequence_Field ***fields**

Pointer to `NULL`-terminated array with field names of the new type.

int **n_in_sequence**

Number of fields visible to the Python side (if used as tuple).

type **`PyStructSequence_Field`**

Część stabilnego ABI *(w tym wszystkie składniki).* Describes a field of a struct sequence. As a struct sequence is modeled as a tuple, all fields are typed as `PyObject*`. The index in the `fields` array of the `PyStructSequence_Desc` determines which field of the struct sequence is described.

const char ***name**

Name for the field or `NULL` to end the list of named fields, set to `PyStructSequence_UnnamedField` to leave unnamed.

const char *doc

Field docstring or NULL to omit.

const char *const **PyStructSequence_UnnamedField**

Część stabilnego ABI *od wersji 3.11*. Special value for a field name to leave it unnamed.

Zmienione w wersji 3.9: The type was changed from `char *`.

PyObject ***PyStructSequence_New**(*PyTypeObject* *type)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Creates an instance of *type*, which must have been created with `PyStructSequence_NewType()`.

Return NULL with an exception set on failure.

PyObject ***PyStructSequence_GetItem**(*PyObject* *p, *Py_ssize_t* pos)

Wartość zwracana: pożyczona referencja. Część stabilnego ABI. Return the object at position *pos* in the struct sequence pointed to by *p*.

Bounds checking is performed as an assertion if Python is built in debug mode or `with assertions`.

PyObject ***PyStructSequence_GET_ITEM**(*PyObject* *p, *Py_ssize_t* pos)

Wartość zwracana: pożyczona referencja. Alias to `PyStructSequence_GetItem()`.

Zmienione w wersji 3.13: Now implemented as an alias to `PyStructSequence_GetItem()`.

void **PyStructSequence_SetItem**(*PyObject* *p, *Py_ssize_t* pos, *PyObject* *o)

Część stabilnego ABI. Sets the field at index *pos* of the struct sequence *p* to value *o*. Like `PyTuple_SET_ITEM()`, this should only be used to fill in brand new instances.

Bounds checking is performed as an assertion if Python is built in debug mode or `with assertions`.

> **Informacja**
>
> This function „steals” a reference to *o*.

void **PyStructSequence_SET_ITEM**(*PyObject* *p, *Py_ssize_t* *pos, *PyObject* *o)

Alias to `PyStructSequence_SetItem()`.

Zmienione w wersji 3.13: Now implemented as an alias to `PyStructSequence_SetItem()`.

8.3.6 List Objects

type **PyListObject**

Ten podtyp `PyObject` reprezentuje obiekt listy Pythona.

PyTypeObject **PyList_Type**

Część stabilnego ABI. This instance of `PyTypeObject` represents the Python list type. This is the same object as `list` in the Python layer.

int **PyList_Check**(*PyObject* *p)

Return true if *p* is a list object or an instance of a subtype of the list type. This function always succeeds.

int **PyList_CheckExact**(*PyObject* *p)

Return true if *p* is a list object, but not an instance of a subtype of the list type. This function always succeeds.

PyObject ***PyList_New**(*Py_ssize_t* len)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return a new list of length *len* on success, or NULL on failure.

> **Informacja**
>
> If *len* is greater than zero, the returned list object's items are set to `NULL`. Thus you cannot use abstract API functions such as `PySequence_SetItem()` or expose the object to Python code before setting all items to a real object with `PyList_SetItem()` or `PyList_SET_ITEM()`. The following APIs are safe APIs before the list is fully initialized: `PyList_SetItem()` and `PyList_SET_ITEM()`.

Py_ssize_t **PyList_Size**(*PyObject* *list)

Część stabilnego ABI. Return the length of the list object in *list*; this is equivalent to `len(list)` on a list object.

Py_ssize_t **PyList_GET_SIZE**(*PyObject* *list)

Similar to `PyList_Size()`, but without error checking.

PyObject ***PyList_GetItemRef**(*PyObject* *list, *Py_ssize_t* index)

Wartość zwracana: nowa referencja. Część stabilnego ABI *od wersji 3.13.* Return the object at position *index* in the list pointed to by *list*. The position must be non-negative; indexing from the end of the list is not supported. If *index* is out of bounds (`<0 or >=len(list)`), return `NULL` and set an `IndexError` exception.

Added in version 3.13.

PyObject ***PyList_GetItem**(*PyObject* *list, *Py_ssize_t* index)

Wartość zwracana: pożyczona referencja. Część stabilnego ABI. Like `PyList_GetItemRef()`, but returns a *borrowed reference* instead of a *strong reference*.

PyObject ***PyList_GET_ITEM**(*PyObject* *list, *Py_ssize_t* i)

Wartość zwracana: pożyczona referencja. Similar to `PyList_GetItem()`, but without error checking.

int **PyList_SetItem**(*PyObject* *list, *Py_ssize_t* index, *PyObject* *item)

Część stabilnego ABI. Set the item at index *index* in list to *item*. Return `0` on success. If *index* is out of bounds, return `-1` and set an `IndexError` exception.

> **Informacja**
>
> This function „steals" a reference to *item* and discards a reference to an item already in the list at the affected position.

void **PyList_SET_ITEM**(*PyObject* *list, *Py_ssize_t* i, *PyObject* *o)

Macro form of `PyList_SetItem()` without error checking. This is normally only used to fill in new lists where there is no previous content.

Bounds checking is performed as an assertion if Python is built in debug mode or `with assertions`.

> **Informacja**
>
> This macro „steals" a reference to *item*, and, unlike `PyList_SetItem()`, does *not* discard a reference to any item that is being replaced; any reference in *list* at position *i* will be leaked.

int **PyList_Insert**(*PyObject* *list, *Py_ssize_t* index, *PyObject* *item)

Część stabilnego ABI. Insert the item *item* into list *list* in front of index *index*. Return `0` if successful; return `-1` and set an exception if unsuccessful. Analogous to `list.insert(index, item)`.

int **PyList_Append**(*PyObject* *list, *PyObject* *item)

Część stabilnego ABI. Append the object *item* at the end of list *list*. Return `0` if successful; return `-1` and set an exception if unsuccessful. Analogous to `list.append(item)`.

PyObject ***PyList_GetSlice*** (*PyObject* *list, *Py_ssize_t* low, *Py_ssize_t* high)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return a list of the objects in *list* containing the objects *between low* and *high*. Return `NULL` and set an exception if unsuccessful. Analogous to `list[low:high]`. Indexing from the end of the list is not supported.

int ***PyList_SetSlice*** (*PyObject* *list, *Py_ssize_t* low, *Py_ssize_t* high, *PyObject* *itemlist)

Część stabilnego ABI. Set the slice of *list* between *low* and *high* to the contents of *itemlist*. Analogous to `list[low:high] = itemlist`. The *itemlist* may be `NULL`, indicating the assignment of an empty list (slice deletion). Return `0` on success, `-1` on failure. Indexing from the end of the list is not supported.

int ***PyList_Extend*** (*PyObject* *list, *PyObject* *iterable)

Extend *list* with the contents of *iterable*. This is the same as `PyList_SetSlice(list, PY_SSIZE_T_MAX, PY_SSIZE_T_MAX, iterable)` and analogous to `list.extend(iterable)` or `list += iterable`.

Raise an exception and return `-1` if *list* is not a `list` object. Return 0 on success.

Added in version 3.13.

int ***PyList_Clear*** (*PyObject* *list)

Remove all items from *list*. This is the same as `PyList_SetSlice(list, 0, PY_SSIZE_T_MAX, NULL)` and analogous to `list.clear()` or `del list[:]`.

Raise an exception and return `-1` if *list* is not a `list` object. Return 0 on success.

Added in version 3.13.

int ***PyList_Sort*** (*PyObject* *list)

Część stabilnego ABI. Sort the items of *list* in place. Return `0` on success, `-1` on failure. This is equivalent to `list.sort()`.

int ***PyList_Reverse*** (*PyObject* *list)

Część stabilnego ABI. Reverse the items of *list* in place. Return `0` on success, `-1` on failure. This is the equivalent of `list.reverse()`.

PyObject ***PyList_AsTuple*** (*PyObject* *list)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return a new tuple object containing the contents of *list*; equivalent to `tuple(list)`.

8.4 Container Objects

8.4.1 Obiekty słownika

type ***PyDictObject***

Ten podtyp `PyObject` reprezentuje obiekt słownika Pythona.

PyTypeObject ***PyDict_Type***

Część stabilnego ABI. Ta instancja `PyTypeObject` reprezentuje typ słownika Pythona. Jest to ten sam obiekt, co `dict` w warstwie Pythona.

int ***PyDict_Check*** (*PyObject* *p)

Zwraca wartość true, jeśli *p* jest obiektem typu dict lub instancją podtypu typu dict. Funkcja ta zawsze kończy się powodzeniem.

int ***PyDict_CheckExact*** (*PyObject* *p)

Zwraca wartość true, jeśli *p* jest obiektem typu dict, ale nie jest instancją podtypu typu dict. Funkcja ta zawsze kończy się powodzeniem.

PyObject ***PyDict_New*** ()

Wartość zwracana: nowa referencja. Część stabilnego ABI. Zwraca nowy pusty słownik lub `NULL` w przypadku niepowodzenia.

PyObject ***PyDictProxy_New**(*PyObject* *mapping)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return a `types.MappingProxyType` object for a mapping which enforces read-only behavior. This is normally used to create a view to prevent modification of the dictionary for non-dynamic class types.

void **PyDict_Clear**(*PyObject* *p)

Część stabilnego ABI. Empty an existing dictionary of all key-value pairs.

int **PyDict_Contains**(*PyObject* *p, *PyObject* *key)

Część stabilnego ABI. Determine if dictionary *p* contains *key*. If an item in *p* is matches *key*, return `1`, otherwise return `0`. On error, return `-1`. This is equivalent to the Python expression `key in p`.

int **PyDict_ContainsString**(*PyObject* *p, const char *key)

This is the same as `PyDict_Contains()`, but *key* is specified as a `const char*` UTF-8 encoded bytes string, rather than a `PyObject*`.

Added in version 3.13.

PyObject ***PyDict_Copy**(*PyObject* *p)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Zwraca nowy słownik zawierający te same pary klucz-wartość co *p*.

int **PyDict_SetItem**(*PyObject* *p, *PyObject* *key, *PyObject* *val)

Część stabilnego ABI. Insert *val* into the dictionary *p* with a key of *key*. *key* must be *hashable*; if it isn't, `TypeError` will be raised. Return `0` on success or `-1` on failure. This function *does not* steal a reference to *val*.

int **PyDict_SetItemString**(*PyObject* *p, const char *key, *PyObject* *val)

Część stabilnego ABI. This is the same as `PyDict_SetItem()`, but *key* is specified as a `const char*` UTF-8 encoded bytes string, rather than a `PyObject*`.

int **PyDict_DelItem**(*PyObject* *p, *PyObject* *key)

Część stabilnego ABI. Remove the entry in dictionary *p* with key *key*. *key* must be *hashable*; if it isn't, `TypeError` is raised. If *key* is not in the dictionary, `KeyError` is raised. Return `0` on success or `-1` on failure.

int **PyDict_DelItemString**(*PyObject* *p, const char *key)

Część stabilnego ABI. This is the same as `PyDict_DelItem()`, but *key* is specified as a `const char*` UTF-8 encoded bytes string, rather than a `PyObject*`.

int **PyDict_GetItemRef**(*PyObject* *p, *PyObject* *key, *PyObject* **result)

Część stabilnego ABI *od wersji 3.13*. Return a new *strong reference* to the object from dictionary *p* which has a key *key*:

- If the key is present, set **result* to a new *strong reference* to the value and return `1`.
- If the key is missing, set **result* to `NULL` and return `0`.
- On error, raise an exception and return `-1`.

Added in version 3.13.

See also the `PyObject_GetItem()` function.

PyObject ***PyDict_GetItem**(*PyObject* *p, *PyObject* *key)

Wartość zwracana: pożyczona referencja. Część stabilnego ABI. Return a *borrowed reference* to the object from dictionary *p* which has a key *key*. Return `NULL` if the key *key* is missing *without* setting an exception.

> **Informacja**
>
> Exceptions that occur while this calls `__hash__()` and `__eq__()` methods are silently ignored. Prefer the `PyDict_GetItemWithError()` function instead.

Zmienione w wersji 3.10: Calling this API without *GIL* held had been allowed for historical reason. It is no longer allowed.

PyObject ***PyDict_GetItemWithError**(*PyObject* *p, *PyObject* *key)

Wartość zwracana: pożyczona referencja. Część stabilnego ABI. Variant of `PyDict_GetItem()` that does not suppress exceptions. Return `NULL` **with** an exception set if an exception occurred. Return `NULL` **without** an exception set if the key wasn't present.

PyObject ***PyDict_GetItemString**(*PyObject* *p, const char *key)

Wartość zwracana: pożyczona referencja. Część stabilnego ABI. This is the same as `PyDict_GetItem()`, but *key* is specified as a `const char*` UTF-8 encoded bytes string, rather than a `PyObject*`.

> **Informacja**
>
> Exceptions that occur while this calls `__hash__()` and `__eq__()` methods or while creating the temporary `str` object are silently ignored. Prefer using the `PyDict_GetItemWithError()` function with your own `PyUnicode_FromString()` *key* instead.

int **PyDict_GetItemStringRef**(*PyObject* *p, const char *key, *PyObject* **result)

Część stabilnego ABI *od wersji 3.13.* Similar than `PyDict_GetItemRef()`, but *key* is specified as a `const char*` UTF-8 encoded bytes string, rather than a `PyObject*`.

Added in version 3.13.

PyObject ***PyDict_SetDefault**(*PyObject* *p, *PyObject* *key, *PyObject* *defaultobj)

Wartość zwracana: pożyczona referencja. This is the same as the Python-level `dict.setdefault()`. If present, it returns the value corresponding to *key* from the dictionary *p*. If the key is not in the dict, it is inserted with value *defaultobj* and *defaultobj* is returned. This function evaluates the hash function of *key* only once, instead of evaluating it independently for the lookup and the insertion.

Added in version 3.4.

int **PyDict_SetDefaultRef**(*PyObject* *p, *PyObject* *key, *PyObject* *default_value, *PyObject* **result)

Inserts *default_value* into the dictionary *p* with a key of *key* if the key is not already present in the dictionary. If *result* is not `NULL`, then **result* is set to a *strong reference* to either *default_value*, if the key was not present, or the existing value, if *key* was already present in the dictionary. Returns `1` if the key was present and *default_value* was not inserted, or `0` if the key was not present and *default_value* was inserted. On failure, returns `-1`, sets an exception, and sets `*result` to `NULL`.

For clarity: if you have a strong reference to *default_value* before calling this function, then after it returns, you hold a strong reference to both *default_value* and **result* (if it's not `NULL`). These may refer to the same object: in that case you hold two separate references to it.

Added in version 3.13.

int **PyDict_Pop**(*PyObject* *p, *PyObject* *key, *PyObject* **result)

Remove *key* from dictionary *p* and optionally return the removed value. Do not raise `KeyError` if the key missing.

- If the key is present, set **result* to a new reference to the removed value if *result* is not `NULL`, and return `1`.
- If the key is missing, set **result* to `NULL` if *result* is not `NULL`, and return `0`.
- On error, raise an exception and return `-1`.

This is similar to `dict.pop()`, but without the default value and not raising `KeyError` if the key missing.

Added in version 3.13.

int **PyDict_PopString**(*PyObject* *p, const char *key, *PyObject* **result)

Similar to `PyDict_Pop()`, but *key* is specified as a `const char*` UTF-8 encoded bytes string, rather than a `PyObject*`.

Added in version 3.13.

PyObject ***PyDict_Items*** (*PyObject* *p)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return a `PyListObject` containing all the items from the dictionary.

PyObject ***PyDict_Keys*** (*PyObject* *p)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return a `PyListObject` containing all the keys from the dictionary.

PyObject ***PyDict_Values*** (*PyObject* *p)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Zwraca `PyListObject` zawierający wszystkie wartości ze słownika *p*.

Py_ssize_t ***PyDict_Size*** (*PyObject* *p)

Część stabilnego ABI. Return the number of items in the dictionary. This is equivalent to `len(p)` on a dictionary.

int ***PyDict_Next*** (*PyObject* *p, *Py_ssize_t* *ppos, *PyObject* **pkey, *PyObject* **pvalue)

Część stabilnego ABI. Iterate over all key-value pairs in the dictionary *p*. The `Py_ssize_t` referred to by *ppos* must be initialized to `0` prior to the first call to this function to start the iteration; the function returns true for each pair in the dictionary, and false once all pairs have been reported. The parameters *pkey* and *pvalue* should either point to `PyObject*` variables that will be filled in with each key and value, respectively, or may be `NULL`. Any references returned through them are borrowed. *ppos* should not be altered during iteration. Its value represents offsets within the internal dictionary structure, and since the structure is sparse, the offsets are not consecutive.

Dla przykładu:

```
PyObject *key, *value;
Py_ssize_t pos = 0;

while (PyDict_Next(self->dict, &pos, &key, &value)) {
    /* do something interesting with the values... */
    ...
}
```

The dictionary *p* should not be mutated during iteration. It is safe to modify the values of the keys as you iterate over the dictionary, but only so long as the set of keys does not change. For example:

```
PyObject *key, *value;
Py_ssize_t pos = 0;

while (PyDict_Next(self->dict, &pos, &key, &value)) {
    long i = PyLong_AsLong(value);
    if (i == -1 && PyErr_Occurred()) {
        return -1;
    }
    PyObject *o = PyLong_FromLong(i + 1);
    if (o == NULL)
        return -1;
    if (PyDict_SetItem(self->dict, key, o) < 0) {
        Py_DECREF(o);
        return -1;
    }
    Py_DECREF(o);
}
```

The function is not thread-safe in the *free-threaded* build without external synchronization. You can use `Py_BEGIN_CRITICAL_SECTION` to lock the dictionary while iterating over it:

```
Py_BEGIN_CRITICAL_SECTION(self->dict);
while (PyDict_Next(self->dict, &pos, &key, &value)) {
    ...
}
Py_END_CRITICAL_SECTION();
```

int **PyDict_Merge**(*PyObject* *a, *PyObject* *b, int override)

Część stabilnego ABI. Iterate over mapping object *b* adding key-value pairs to dictionary *a*. *b* may be a dictionary, or any object supporting `PyMapping_Keys()` and `PyObject_GetItem()`. If *override* is true, existing pairs in *a* will be replaced if a matching key is found in *b*, otherwise pairs will only be added if there is not a matching key in *a*. Return `0` on success or `-1` if an exception was raised.

int **PyDict_Update**(*PyObject* *a, *PyObject* *b)

Część stabilnego ABI. This is the same as `PyDict_Merge(a, b, 1)` in C, and is similar to `a.update(b)` in Python except that `PyDict_Update()` doesn't fall back to the iterating over a sequence of key value pairs if the second argument has no „keys" attribute. Return `0` on success or `-1` if an exception was raised.

int **PyDict_MergeFromSeq2**(*PyObject* *a, *PyObject* *seq2, int override)

Część stabilnego ABI. Update or merge into dictionary *a*, from the key-value pairs in *seq2*. *seq2* must be an iterable object producing iterable objects of length 2, viewed as key-value pairs. In case of duplicate keys, the last wins if *override* is true, else the first wins. Return `0` on success or `-1` if an exception was raised. Equivalent Python (except for the return value):

```
def PyDict_MergeFromSeq2(a, seq2, override):
    for key, value in seq2:
        if override or key not in a:
            a[key] = value
```

int **PyDict_AddWatcher**(*PyDict_WatchCallback* callback)

Register *callback* as a dictionary watcher. Return a non-negative integer id which must be passed to future calls to `PyDict_Watch()`. In case of error (e.g. no more watcher IDs available), return `-1` and set an exception.

Added in version 3.12.

int **PyDict_ClearWatcher**(int watcher_id)

Clear watcher identified by *watcher_id* previously returned from `PyDict_AddWatcher()`. Return `0` on success, `-1` on error (e.g. if the given *watcher_id* was never registered.)

Added in version 3.12.

int **PyDict_Watch**(int watcher_id, *PyObject* *dict)

Mark dictionary *dict* as watched. The callback granted *watcher_id* by `PyDict_AddWatcher()` will be called when *dict* is modified or deallocated. Return `0` on success or `-1` on error.

Added in version 3.12.

int **PyDict_Unwatch**(int watcher_id, *PyObject* *dict)

Mark dictionary *dict* as no longer watched. The callback granted *watcher_id* by `PyDict_AddWatcher()` will no longer be called when *dict* is modified or deallocated. The dict must previously have been watched by this watcher. Return `0` on success or `-1` on error.

Added in version 3.12.

type **PyDict_WatchEvent**

Enumeration of possible dictionary watcher events: `PyDict_EVENT_ADDED`, `PyDict_EVENT_MODIFIED`, `PyDict_EVENT_DELETED`, `PyDict_EVENT_CLONED`, `PyDict_EVENT_CLEARED`, or `PyDict_EVENT_DEALLOCATED`.

Added in version 3.12.

typedef int (*`PyDict_WatchCallback`)(*PyDict_WatchEvent* event, *PyObject* *dict, *PyObject* *key, *PyObject* *new_value)

Type of a dict watcher callback function.

If *event* is `PyDict_EVENT_CLEARED` or `PyDict_EVENT_DEALLOCATED`, both *key* and *new_value* will be `NULL`. If *event* is `PyDict_EVENT_ADDED` or `PyDict_EVENT_MODIFIED`, *new_value* will be the new value for *key*. If *event* is `PyDict_EVENT_DELETED`, *key* is being deleted from the dictionary and *new_value* will be `NULL`.

`PyDict_EVENT_CLONED` occurs when *dict* was previously empty and another dict is merged into it. To maintain efficiency of this operation, per-key `PyDict_EVENT_ADDED` events are not issued in this case; instead a single `PyDict_EVENT_CLONED` is issued, and *key* will be the source dictionary.

The callback may inspect but must not modify *dict*; doing so could have unpredictable effects, including infinite recursion. Do not trigger Python code execution in the callback, as it could modify the dict as a side effect.

If *event* is `PyDict_EVENT_DEALLOCATED`, taking a new reference in the callback to the about-to-be-destroyed dictionary will resurrect it and prevent it from being freed at this time. When the resurrected object is destroyed later, any watcher callbacks active at that time will be called again.

Callbacks occur before the notified modification to *dict* takes place, so the prior state of *dict* can be inspected.

If the callback sets an exception, it must return `-1`; this exception will be printed as an unraisable exception using `PyErr_WriteUnraisable()`. Otherwise it should return `0`.

There may already be a pending exception set on entry to the callback. In this case, the callback should return `0` with the same exception still set. This means the callback may not call any other API that can set an exception unless it saves and clears the exception state first, and restores it before returning.

Added in version 3.12.

8.4.2 Set Objects

This section details the public API for `set` and `frozenset` objects. Any functionality not listed below is best accessed using either the abstract object protocol (including `PyObject_CallMethod()`, `PyObject_RichCompareBool()`, `PyObject_Hash()`, `PyObject_Repr()`, `PyObject_IsTrue()`, `PyObject_Print()`, and `PyObject_GetIter()`) or the abstract number protocol (including `PyNumber_And()`, `PyNumber_Subtract()`, `PyNumber_Or()`, `PyNumber_Xor()`, `PyNumber_InPlaceAnd()`, `PyNumber_InPlaceSubtract()`, `PyNumber_InPlaceOr()`, and `PyNumber_InPlaceXor()`).

type **`PySetObject`**

This subtype of `PyObject` is used to hold the internal data for both `set` and `frozenset` objects. It is like a `PyDictObject` in that it is a fixed size for small sets (much like tuple storage) and will point to a separate, variable sized block of memory for medium and large sized sets (much like list storage). None of the fields of this structure should be considered public and all are subject to change. All access should be done through the documented API rather than by manipulating the values in the structure.

PyTypeObject **`PySet_Type`**

Część stabilnego ABI. This is an instance of `PyTypeObject` representing the Python `set` type.

PyTypeObject **`PyFrozenSet_Type`**

Część stabilnego ABI. This is an instance of `PyTypeObject` representing the Python `frozenset` type.

The following type check macros work on pointers to any Python object. Likewise, the constructor functions work with any iterable Python object.

int **`PySet_Check`**(*PyObject* *p)

Return true if *p* is a `set` object or an instance of a subtype. This function always succeeds.

int **`PyFrozenSet_Check`**(*PyObject* *p)

Return true if *p* is a `frozenset` object or an instance of a subtype. This function always succeeds.

int **`PyAnySet_Check`**(*PyObject* *p)

Return true if *p* is a `set` object, a `frozenset` object, or an instance of a subtype. This function always succeeds.

int **PySet_CheckExact**(*PyObject* *p)

Return true if *p* is a `set` object but not an instance of a subtype. This function always succeeds.

Added in version 3.10.

int **PyAnySet_CheckExact**(*PyObject* *p)

Return true if *p* is a `set` object or a `frozenset` object but not an instance of a subtype. This function always succeeds.

int **PyFrozenSet_CheckExact**(*PyObject* *p)

Return true if *p* is a `frozenset` object but not an instance of a subtype. This function always succeeds.

PyObject ***PySet_New**(*PyObject* *iterable)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return a new `set` containing objects returned by the *iterable*. The *iterable* may be `NULL` to create a new empty set. Return the new set on success or `NULL` on failure. Raise `TypeError` if *iterable* is not actually iterable. The constructor is also useful for copying a set (`c=set(s)`).

PyObject ***PyFrozenSet_New**(*PyObject* *iterable)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return a new `frozenset` containing objects returned by the *iterable*. The *iterable* may be `NULL` to create a new empty frozenset. Return the new set on success or `NULL` on failure. Raise `TypeError` if *iterable* is not actually iterable.

The following functions and macros are available for instances of `set` or `frozenset` or instances of their subtypes.

Py_ssize_t **PySet_Size**(*PyObject* *anyset)

Część stabilnego ABI. Return the length of a `set` or `frozenset` object. Equivalent to `len(anyset)`. Raises a `SystemError` if *anyset* is not a `set`, `frozenset`, or an instance of a subtype.

Py_ssize_t **PySet_GET_SIZE**(*PyObject* *anyset)

Macro form of `PySet_Size()` without error checking.

int **PySet_Contains**(*PyObject* *anyset, *PyObject* *key)

Część stabilnego ABI. Return `1` if found, `0` if not found, and `-1` if an error is encountered. Unlike the Python `__contains__()` method, this function does not automatically convert unhashable sets into temporary frozensets. Raise a `TypeError` if the *key* is unhashable. Raise `SystemError` if *anyset* is not a `set`, `frozenset`, or an instance of a subtype.

int **PySet_Add**(*PyObject* *set, *PyObject* *key)

Część stabilnego ABI. Add *key* to a `set` instance. Also works with `frozenset` instances (like `PyTuple_SetItem()` it can be used to fill in the values of brand new frozensets before they are exposed to other code). Return `0` on success or `-1` on failure. Raise a `TypeError` if the *key* is unhashable. Raise a `MemoryError` if there is no room to grow. Raise a `SystemError` if *set* is not an instance of `set` or its subtype.

The following functions are available for instances of `set` or its subtypes but not for instances of `frozenset` or its subtypes.

int **PySet_Discard**(*PyObject* *set, *PyObject* *key)

Część stabilnego ABI. Return `1` if found and removed, `0` if not found (no action taken), and `-1` if an error is encountered. Does not raise `KeyError` for missing keys. Raise a `TypeError` if the *key* is unhashable. Unlike the Python `discard()` method, this function does not automatically convert unhashable sets into temporary frozensets. Raise `SystemError` if *set* is not an instance of `set` or its subtype.

PyObject ***PySet_Pop**(*PyObject* *set)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return a new reference to an arbitrary object in the *set*, and removes the object from the *set*. Return `NULL` on failure. Raise `KeyError` if the set is empty. Raise a `SystemError` if *set* is not an instance of `set` or its subtype.

int **PySet_Clear**(*PyObject* *set)

Część stabilnego ABI. Empty an existing set of all elements. Return `0` on success. Return `-1` and raise `SystemError` if *set* is not an instance of `set` or its subtype.

8.5 Obiekty Funkcja

8.5.1 Obiekty Funkcja

Istnieje kilka funkcji specyficznych dla funkcji Pythona.

type **PyFunctionObject**

Struktura C używana dla funkcji.

PyTypeObject **PyFunction_Type**

This is an instance of `PyTypeObject` and represents the Python function type. It is exposed to Python programmers as `types.FunctionType`.

int **PyFunction_Check**(*PyObject* *o)

Return true if *o* is a function object (has type `PyFunction_Type`). The parameter must not be `NULL`. This function always succeeds.

PyObject ***PyFunction_New**(*PyObject* *code, *PyObject* *globals)

Wartość zwracana: nowa referencja. Return a new function object associated with the code object *code*. *globals* must be a dictionary with the global variables accessible to the function.

The function's docstring and name are retrieved from the code object. `__module__` is retrieved from *globals*. The argument defaults, annotations and closure are set to `NULL`. `__qualname__` is set to the same value as the code object's `co_qualname` field.

PyObject ***PyFunction_NewWithQualName**(*PyObject* *code, *PyObject* *globals, *PyObject* *qualname)

Wartość zwracana: nowa referencja. As `PyFunction_New()`, but also allows setting the function object's `__qualname__` attribute. *qualname* should be a unicode object or `NULL`; if `NULL`, the `__qualname__` attribute is set to the same value as the code object's `co_qualname` field.

Added in version 3.3.

PyObject ***PyFunction_GetCode**(*PyObject* *op)

Wartość zwracana: pożyczona referencja. Return the code object associated with the function object *op*.

PyObject ***PyFunction_GetGlobals**(*PyObject* *op)

Wartość zwracana: pożyczona referencja. Return the globals dictionary associated with the function object *op*.

PyObject ***PyFunction_GetModule**(*PyObject* *op)

Wartość zwracana: pożyczona referencja. Return a *borrowed reference* to the `__module__` attribute of the function object *op*. It can be *NULL*.

This is normally a `string` containing the module name, but can be set to any other object by Python code.

PyObject ***PyFunction_GetDefaults**(*PyObject* *op)

Wartość zwracana: pożyczona referencja. Return the argument default values of the function object *op*. This can be a tuple of arguments or `NULL`.

int **PyFunction_SetDefaults**(*PyObject* *op, *PyObject* *defaults)

Set the argument default values for the function object *op*. *defaults* must be `Py_None` or a tuple.

Raises `SystemError` and returns `-1` on failure.

void **PyFunction_SetVectorcall**(*PyFunctionObject* *func, *vectorcallfunc* vectorcall)

Set the vectorcall field of a given function object *func*.

Warning: extensions using this API must preserve the behavior of the unaltered (default) vectorcall function!

Added in version 3.12.

PyObject ***PyFunction_GetClosure**(*PyObject* *op)

Wartość zwracana: pożyczona referencja. Return the closure associated with the function object *op*. This can be `NULL` or a tuple of cell objects.

int **PyFunction_SetClosure**(*PyObject* *op, *PyObject* *closure)

Set the closure associated with the function object *op*. *closure* must be `Py_None` or a tuple of cell objects.

Raises `SystemError` and returns `-1` on failure.

PyObject ***PyFunction_GetAnnotations**(*PyObject* *op)

Wartość zwracana: pożyczona referencja. Return the annotations of the function object *op*. This can be a mutable dictionary or `NULL`.

int **PyFunction_SetAnnotations**(*PyObject* *op, *PyObject* *annotations)

Set the annotations for the function object *op*. *annotations* must be a dictionary or `Py_None`.

Raises `SystemError` and returns `-1` on failure.

int **PyFunction_AddWatcher**(*PyFunction_WatchCallback* callback)

Register *callback* as a function watcher for the current interpreter. Return an ID which may be passed to `PyFunction_ClearWatcher()`. In case of error (e.g. no more watcher IDs available), return `-1` and set an exception.

Added in version 3.12.

int **PyFunction_ClearWatcher**(int watcher_id)

Clear watcher identified by *watcher_id* previously returned from `PyFunction_AddWatcher()` for the current interpreter. Return `0` on success, or `-1` and set an exception on error (e.g. if the given *watcher_id* was never registered.)

Added in version 3.12.

type **PyFunction_WatchEvent**

Enumeration of possible function watcher events: - `PyFunction_EVENT_CREATE` - `PyFunction_EVENT_DESTROY` - `PyFunction_EVENT_MODIFY_CODE` - `PyFunction_EVENT_MODIFY_DEFAULTS` - `PyFunction_EVENT_MODIFY_KWDEFAULTS`

Added in version 3.12.

typedef int (***PyFunction_WatchCallback**)(*PyFunction_WatchEvent* event, *PyFunctionObject* *func, *PyObject* *new_value)

Type of a function watcher callback function.

If *event* is `PyFunction_EVENT_CREATE` or `PyFunction_EVENT_DESTROY` then *new_value* will be `NULL`. Otherwise, *new_value* will hold a *borrowed reference* to the new value that is about to be stored in *func* for the attribute that is being modified.

The callback may inspect but must not modify *func*; doing so could have unpredictable effects, including infinite recursion.

If *event* is `PyFunction_EVENT_CREATE`, then the callback is invoked after *func* has been fully initialized. Otherwise, the callback is invoked before the modification to *func* takes place, so the prior state of *func* can be inspected. The runtime is permitted to optimize away the creation of function objects when possible. In such cases no event will be emitted. Although this creates the possibility of an observable difference of runtime behavior depending on optimization decisions, it does not change the semantics of the Python code being executed.

If *event* is `PyFunction_EVENT_DESTROY`, Taking a reference in the callback to the about-to-be-destroyed function will resurrect it, preventing it from being freed at this time. When the resurrected object is destroyed later, any watcher callbacks active at that time will be called again.

If the callback sets an exception, it must return `-1`; this exception will be printed as an unraisable exception using `PyErr_WriteUnraisable()`. Otherwise it should return `0`.

There may already be a pending exception set on entry to the callback. In this case, the callback should return `0` with the same exception still set. This means the callback may not call any other API that can set an exception unless it saves and clears the exception state first, and restores it before returning.

Added in version 3.12.

8.5.2 Instance Method Objects

An instance method is a wrapper for a `PyCFunction` and the new way to bind a `PyCFunction` to a class object. It replaces the former call `PyMethod_New(func, NULL, class)`.

PyTypeObject **PyInstanceMethod_Type**

This instance of `PyTypeObject` represents the Python instance method type. It is not exposed to Python programs.

int **PyInstanceMethod_Check**(*PyObject* *o)

Return true if *o* is an instance method object (has type `PyInstanceMethod_Type`). The parameter must not be `NULL`. This function always succeeds.

PyObject ***PyInstanceMethod_New**(*PyObject* *func)

Wartość zwracana: nowa referencja. Return a new instance method object, with *func* being any callable object. *func* is the function that will be called when the instance method is called.

PyObject ***PyInstanceMethod_Function**(*PyObject* *im)

Wartość zwracana: pożyczona referencja. Return the function object associated with the instance method *im*.

PyObject ***PyInstanceMethod_GET_FUNCTION**(*PyObject* *im)

Wartość zwracana: pożyczona referencja. Macro version of `PyInstanceMethod_Function()` which avoids error checking.

8.5.3 Method Objects

Methods are bound function objects. Methods are always bound to an instance of a user-defined class. Unbound methods (methods bound to a class object) are no longer available.

PyTypeObject **PyMethod_Type**

This instance of `PyTypeObject` represents the Python method type. This is exposed to Python programs as `types.MethodType`.

int **PyMethod_Check**(*PyObject* *o)

Return true if *o* is a method object (has type `PyMethod_Type`). The parameter must not be `NULL`. This function always succeeds.

PyObject ***PyMethod_New**(*PyObject* *func, *PyObject* *self)

Wartość zwracana: nowa referencja. Return a new method object, with *func* being any callable object and *self* the instance the method should be bound. *func* is the function that will be called when the method is called. *self* must not be `NULL`.

PyObject ***PyMethod_Function**(*PyObject* *meth)

Wartość zwracana: pożyczona referencja. Return the function object associated with the method *meth*.

PyObject ***PyMethod_GET_FUNCTION**(*PyObject* *meth)

Wartość zwracana: pożyczona referencja. Macro version of `PyMethod_Function()` which avoids error checking.

PyObject ***PyMethod_Self**(*PyObject* *meth)

Wartość zwracana: pożyczona referencja. Return the instance associated with the method *meth*.

PyObject ***PyMethod_GET_SELF**(*PyObject* *meth)

Wartość zwracana: pożyczona referencja. Macro version of `PyMethod_Self()` which avoids error checking.

8.5.4 Cell Objects

Obiekty „Cell” są używane do implementacji zmiennych, do których odwołuje się wiele zakresów. Dla każdej takiej zmiennej tworzony jest obiekt komórki do przechowywania wartości; zmienne lokalne każdej ramki stosu, która odwołuje się do wartości, zawierają odniesienie do komórek z zewnętrznych zakresów, które również używają tej zmiennej. Gdy wartość jest dostępna, wartość zawarta w komórce jest używana zamiast samego obiektu komórki.

Odwoływanie się do obiektu komórki wymaga wsparcia ze strony wygenerowanego kodu bajtowego; nie są one automatycznie odwoływane, gdy uzyskuje się do nich dostęp. Obiekty komórek prawdopodobnie nie będą przydatne w innych miejscach.

type **PyCellObject**

The C structure used for cell objects.

PyTypeObject **PyCell_Type**

The type object corresponding to cell objects.

int **PyCell_Check**(*PyObject* *ob)

Return true if *ob* is a cell object; *ob* must not be `NULL`. This function always succeeds.

PyObject ***PyCell_New**(*PyObject* *ob)

Wartość zwracana: nowa referencja. Create and return a new cell object containing the value *ob*. The parameter may be `NULL`.

PyObject ***PyCell_Get**(*PyObject* *cell)

Wartość zwracana: nowa referencja. Return the contents of the cell *cell*, which can be `NULL`. If *cell* is not a cell object, returns `NULL` with an exception set.

PyObject ***PyCell_GET**(*PyObject* *cell)

Wartość zwracana: pożyczona referencja. Return the contents of the cell *cell*, but without checking that *cell* is non-`NULL` and a cell object.

int **PyCell_Set**(*PyObject* *cell, *PyObject* *value)

Set the contents of the cell object *cell* to *value*. This releases the reference to any current content of the cell. *value* may be `NULL`. *cell* must be non-`NULL`.

On success, return `0`. If *cell* is not a cell object, set an exception and return `-1`.

void **PyCell_SET**(*PyObject* *cell, *PyObject* *value)

Sets the value of the cell object *cell* to *value*. No reference counts are adjusted, and no checks are made for safety; *cell* must be non-`NULL` and must be a cell object.

8.5.5 Code Objects

Code objects are a low-level detail of the CPython implementation. Each one represents a chunk of executable code that hasn't yet been bound into a function.

type **PyCodeObject**

The C structure of the objects used to describe code objects. The fields of this type are subject to change at any time.

PyTypeObject **PyCode_Type**

This is an instance of `PyTypeObject` representing the Python code object.

int **PyCode_Check**(*PyObject* *co)

Return true if *co* is a code object. This function always succeeds.

Py_ssize_t **PyCode_GetNumFree**(*PyCodeObject* *co)

Return the number of *free (closure) variables* in a code object.

int **PyUnstable_Code_GetFirstFree**(*PyCodeObject* *co)

> To jest *niestabilne API*. Może ulec zmianie bez ostrzeżenia w wydaniach minor.

Return the position of the first *free (closure) variable* in a code object.

Zmienione w wersji 3.13: Renamed from `PyCode_GetFirstFree` as part of *Unstable C API*. The old name is deprecated, but will remain available until the signature changes again.

PyCodeObject ***PyUnstable_Code_New*** (int argcount, int kwonlyargcount, int nlocals, int stacksize, int flags, *PyObject* *code, *PyObject* *consts, *PyObject* *names, *PyObject* *varnames, *PyObject* *freevars, *PyObject* *cellvars, *PyObject* *filename, *PyObject* *name, *PyObject* *qualname, int firstlineno, *PyObject* *linetable, *PyObject* *exceptiontable)

> To jest *niestabilne API*. Może ulec zmianie bez ostrzeżenia w wydaniach minor.

Return a new code object. If you need a dummy code object to create a frame, use `PyCode_NewEmpty()` instead.

Since the definition of the bytecode changes often, calling `PyUnstable_Code_New()` directly can bind you to a precise Python version.

The many arguments of this function are inter-dependent in complex ways, meaning that subtle changes to values are likely to result in incorrect execution or VM crashes. Use this function only with extreme care.

Zmienione w wersji 3.11: Added `qualname` and `exceptiontable` parameters.

Zmienione w wersji 3.12: Renamed from `PyCode_New` as part of *Unstable C API*. The old name is deprecated, but will remain available until the signature changes again.

PyCodeObject ***PyUnstable_Code_NewWithPosOnlyArgs*** (int argcount, int posonlyargcount, int kwonlyargcount, int nlocals, int stacksize, int flags, *PyObject* *code, *PyObject* *consts, *PyObject* *names, *PyObject* *varnames, *PyObject* *freevars, *PyObject* *cellvars, *PyObject* *filename, *PyObject* *name, *PyObject* *qualname, int firstlineno, *PyObject* *linetable, *PyObject* *exceptiontable)

> To jest *niestabilne API*. Może ulec zmianie bez ostrzeżenia w wydaniach minor.

Similar to `PyUnstable_Code_New()`, but with an extra „posonlyargcount” for positional-only arguments. The same caveats that apply to `PyUnstable_Code_New` also apply to this function.

Added in version 3.8: as `PyCode_NewWithPosOnlyArgs`

Zmienione w wersji 3.11: Added `qualname` and `exceptiontable` parameters.

Zmienione w wersji 3.12: Renamed to `PyUnstable_Code_NewWithPosOnlyArgs`. The old name is deprecated, but will remain available until the signature changes again.

PyCodeObject ***PyCode_NewEmpty*** (const char *filename, const char *funcname, int firstlineno)

Wartość zwracana: nowa referencja. Return a new empty code object with the specified filename, function name, and first line number. The resulting code object will raise an `Exception` if executed.

int ***PyCode_Addr2Line*** (*PyCodeObject* *co, int byte_offset)

Return the line number of the instruction that occurs on or before `byte_offset` and ends after it. If you just need the line number of a frame, use `PyFrame_GetLineNumber()` instead.

For efficiently iterating over the line numbers in a code object, use **the API described in PEP 626**.

int ***PyCode_Addr2Location*** (*PyObject* *co, int byte_offset, int *start_line, int *start_column, int *end_line, int *end_column)

Sets the passed `int` pointers to the source code line and column numbers for the instruction at `byte_offset`. Sets the value to `0` when information is not available for any particular element.

Returns `1` if the function succeeds and 0 otherwise.

Added in version 3.11.

PyObject ***PyCode_GetCode**(*PyCodeObject* *co)

Equivalent to the Python code `getattr(co, 'co_code')`. Returns a strong reference to a `PyBytesObject` representing the bytecode in a code object. On error, `NULL` is returned and an exception is raised.

This `PyBytesObject` may be created on-demand by the interpreter and does not necessarily represent the bytecode actually executed by CPython. The primary use case for this function is debuggers and profilers.

Added in version 3.11.

PyObject ***PyCode_GetVarnames**(*PyCodeObject* *co)

Equivalent to the Python code `getattr(co, 'co_varnames')`. Returns a new reference to a `PyTupleObject` containing the names of the local variables. On error, `NULL` is returned and an exception is raised.

Added in version 3.11.

PyObject ***PyCode_GetCellvars**(*PyCodeObject* *co)

Equivalent to the Python code `getattr(co, 'co_cellvars')`. Returns a new reference to a `PyTupleObject` containing the names of the local variables that are referenced by nested functions. On error, `NULL` is returned and an exception is raised.

Added in version 3.11.

PyObject ***PyCode_GetFreevars**(*PyCodeObject* *co)

Equivalent to the Python code `getattr(co, 'co_freevars')`. Returns a new reference to a `PyTupleObject` containing the names of the *free (closure) variables*. On error, `NULL` is returned and an exception is raised.

Added in version 3.11.

int **PyCode_AddWatcher**(*PyCode_WatchCallback* callback)

Register *callback* as a code object watcher for the current interpreter. Return an ID which may be passed to `PyCode_ClearWatcher()`. In case of error (e.g. no more watcher IDs available), return `-1` and set an exception.

Added in version 3.12.

int **PyCode_ClearWatcher**(int watcher_id)

Clear watcher identified by *watcher_id* previously returned from `PyCode_AddWatcher()` for the current interpreter. Return `0` on success, or `-1` and set an exception on error (e.g. if the given *watcher_id* was never registered.)

Added in version 3.12.

type **PyCodeEvent**

Enumeration of possible code object watcher events: - `PY_CODE_EVENT_CREATE` - `PY_CODE_EVENT_DESTROY`

Added in version 3.12.

typedef int (***PyCode_WatchCallback**)(*PyCodeEvent* event, *PyCodeObject* *co)

Type of a code object watcher callback function.

If *event* is `PY_CODE_EVENT_CREATE`, then the callback is invoked after *co* has been fully initialized. Otherwise, the callback is invoked before the destruction of *co* takes place, so the prior state of *co* can be inspected.

If *event* is `PY_CODE_EVENT_DESTROY`, taking a reference in the callback to the about-to-be-destroyed code object will resurrect it and prevent it from being freed at this time. When the resurrected object is destroyed later, any watcher callbacks active at that time will be called again.

Users of this API should not rely on internal runtime implementation details. Such details may include, but are not limited to, the exact order and timing of creation and destruction of code objects. While changes in these details may result in differences observable by watchers (including whether a callback is invoked or not), it does not change the semantics of the Python code being executed.

If the callback sets an exception, it must return `-1`; this exception will be printed as an unraisable exception using `PyErr_WriteUnraisable()`. Otherwise it should return `0`.

There may already be a pending exception set on entry to the callback. In this case, the callback should return `0` with the same exception still set. This means the callback may not call any other API that can set an exception unless it saves and clears the exception state first, and restores it before returning.

Added in version 3.12.

8.5.6 Extra information

To support low-level extensions to frame evaluation, such as external just-in-time compilers, it is possible to attach arbitrary extra data to code objects.

These functions are part of the unstable C API tier: this functionality is a CPython implementation detail, and the API may change without deprecation warnings.

Py_ssize_t **PyUnstable_Eval_RequestCodeExtraIndex**(*freefunc* free)

> To jest *niestabilne API*. Może ulec zmianie bez ostrzeżenia w wydaniach minor.

Return a new an opaque index value used to adding data to code objects.

You generally call this function once (per interpreter) and use the result with `PyCode_GetExtra` and `PyCode_SetExtra` to manipulate data on individual code objects.

If *free* is not `NULL`: when a code object is deallocated, *free* will be called on non-`NULL` data stored under the new index. Use `Py_DecRef()` when storing `PyObject`.

Added in version 3.6: as `_PyEval_RequestCodeExtraIndex`

Zmienione w wersji 3.12: Renamed to `PyUnstable_Eval_RequestCodeExtraIndex`. The old private name is deprecated, but will be available until the API changes.

int **PyUnstable_Code_GetExtra**(*PyObject* *code, *Py_ssize_t* index, void **extra)

> To jest *niestabilne API*. Może ulec zmianie bez ostrzeżenia w wydaniach minor.

Set *extra* to the extra data stored under the given index. Return 0 on success. Set an exception and return -1 on failure.

If no data was set under the index, set *extra* to `NULL` and return 0 without setting an exception.

Added in version 3.6: as `_PyCode_GetExtra`

Zmienione w wersji 3.12: Renamed to `PyUnstable_Code_GetExtra`. The old private name is deprecated, but will be available until the API changes.

int **PyUnstable_Code_SetExtra**(*PyObject* *code, *Py_ssize_t* index, void *extra)

> To jest *niestabilne API*. Może ulec zmianie bez ostrzeżenia w wydaniach minor.

Set the extra data stored under the given index to *extra*. Return 0 on success. Set an exception and return -1 on failure.

Added in version 3.6: as `_PyCode_SetExtra`

Zmienione w wersji 3.12: Renamed to `PyUnstable_Code_SetExtra`. The old private name is deprecated, but will be available until the API changes.

8.6 Other Objects

8.6.1 File Objects

These APIs are a minimal emulation of the Python 2 C API for built-in file objects, which used to rely on the buffered I/O (`FILE*`) support from the C standard library. In Python 3, files and streams use the new `io` module, which defines several layers over the low-level unbuffered I/O of the operating system. The functions described below are convenience C wrappers over these new APIs, and meant mostly for internal error reporting in the interpreter; third-party code is advised to access the `io` APIs instead.

PyObject ***PyFile_FromFd**(int fd, const char *name, const char *mode, int buffering, const char *encoding, const char *errors, const char *newline, int closefd)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Create a Python file object from the file descriptor of an already opened file *fd*. The arguments *name*, *encoding*, *errors* and *newline* can be `NULL` to use the defaults; *buffering* can be *-1* to use the default. *name* is ignored and kept for backward compatibility. Return `NULL` on failure. For a more comprehensive description of the arguments, please refer to the `io.open()` function documentation.

> **Ostrzeżenie**
>
> Since Python streams have their own buffering layer, mixing them with OS-level file descriptors can produce various issues (such as unexpected ordering of data).

Zmienione w wersji 3.2: Zignoruj atrybut *name*.

int **PyObject_AsFileDescriptor**(*PyObject* *p)

Część stabilnego ABI. Return the file descriptor associated with *p* as an `int`. If the object is an integer, its value is returned. If not, the object's `fileno()` method is called if it exists; the method must return an integer, which is returned as the file descriptor value. Sets an exception and returns `-1` on failure.

PyObject ***PyFile_GetLine**(*PyObject* *p, int n)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Equivalent to `p.readline([n])`, this function reads one line from the object *p*. *p* may be a file object or any object with a `readline()` method. If *n* is `0`, exactly one line is read, regardless of the length of the line. If *n* is greater than `0`, no more than *n* bytes will be read from the file; a partial line can be returned. In both cases, an empty string is returned if the end of the file is reached immediately. If *n* is less than `0`, however, one line is read regardless of length, but `EOFError` is raised if the end of the file is reached immediately.

int **PyFile_SetOpenCodeHook**(*Py_OpenCodeHookFunction* handler)

Overrides the normal behavior of `io.open_code()` to pass its parameter through the provided handler.

The *handler* is a function of type:

typedef *PyObject* *(***Py_OpenCodeHookFunction**)(*PyObject**, void*)

Equivalent of `PyObject *(*)(PyObject *path, void *userData)`, where *path* is guaranteed to be `PyUnicodeObject`.

The *userData* pointer is passed into the hook function. Since hook functions may be called from different runtimes, this pointer should not refer directly to Python state.

As this hook is intentionally used during import, avoid importing new modules during its execution unless they are known to be frozen or available in `sys.modules`.

Once a hook has been set, it cannot be removed or replaced, and later calls to `PyFile_SetOpenCodeHook()` will fail. On failure, the function returns -1 and sets an exception if the interpreter has been initialized.

Funkcja ta może być bezpiecznie wywołana przed `Py_Initialize()`.

Raises an auditing event `setopencodehook` with no arguments.

Added in version 3.8.

int **PyFile_WriteObject**(*PyObject* *obj, *PyObject* *p, int flags)

Część stabilnego ABI. Write object *obj* to file object *p*. The only supported flag for *flags* is `Py_PRINT_RAW`; if given, the `str()` of the object is written instead of the `repr()`. Return `0` on success or `-1` on failure; the appropriate exception will be set.

int **PyFile_WriteString**(const char *s, *PyObject* *p)

Część stabilnego ABI. Write string *s* to file object *p*. Return `0` on success or `-1` on failure; the appropriate exception will be set.

8.6.2 Module Objects

PyTypeObject **PyModule_Type**

Część stabilnego ABI. This instance of `PyTypeObject` represents the Python module type. This is exposed to Python programs as `types.ModuleType`.

int **PyModule_Check**(*PyObject* *p)

Return true if *p* is a module object, or a subtype of a module object. This function always succeeds.

int **PyModule_CheckExact**(*PyObject* *p)

Return true if *p* is a module object, but not a subtype of `PyModule_Type`. This function always succeeds.

PyObject ***PyModule_NewObject**(*PyObject* *name)

Wartość zwracana: nowa referencja. Część stabilnego ABI *od wersji 3.7.* Return a new module object with `module.__name__` set to *name*. The module's `__name__`, `__doc__`, `__package__` and `__loader__` attributes are filled in (all but `__name__` are set to `None`). The caller is responsible for setting a `__file__` attribute.

Return `NULL` with an exception set on error.

Added in version 3.3.

Zmienione w wersji 3.4: `__package__` and `__loader__` are now set to `None`.

PyObject ***PyModule_New**(const char *name)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Similar to `PyModule_NewObject()`, but the name is a UTF-8 encoded string instead of a Unicode object.

PyObject ***PyModule_GetDict**(*PyObject* *module)

Wartość zwracana: pożyczona referencja. Część stabilnego ABI. Return the dictionary object that implements *module*'s namespace; this object is the same as the `__dict__` attribute of the module object. If *module* is not a module object (or a subtype of a module object), `SystemError` is raised and `NULL` is returned.

It is recommended extensions use other `PyModule_*` and `PyObject_*` functions rather than directly manipulate a module's `__dict__`.

PyObject ***PyModule_GetNameObject**(*PyObject* *module)

Wartość zwracana: nowa referencja. Część stabilnego ABI *od wersji 3.7.* Return *module*'s `__name__` value. If the module does not provide one, or if it is not a string, `SystemError` is raised and `NULL` is returned.

Added in version 3.3.

const char ***PyModule_GetName**(*PyObject* *module)

Część stabilnego ABI. Similar to `PyModule_GetNameObject()` but return the name encoded to `'utf-8'`.

void ***PyModule_GetState**(*PyObject* *module)

Część stabilnego ABI. Return the „state" of the module, that is, a pointer to the block of memory allocated at module creation time, or `NULL`. See `PyModuleDef.m_size`.

PyModuleDef ***PyModule_GetDef**(*PyObject* *module)

Część stabilnego ABI. Return a pointer to the `PyModuleDef` struct from which the module was created, or `NULL` if the module wasn't created from a definition.

PyObject ***PyModule_GetFilenameObject**(*PyObject* *module)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return the name of the file from which *module* was loaded using *module*'s `__file__` attribute. If this is not defined, or if it is not a string, raise `SystemError` and return `NULL`; otherwise return a reference to a Unicode object.

Added in version 3.2.

const char ***PyModule_GetFilename**(*PyObject* *module)

Część stabilnego ABI. Similar to `PyModule_GetFilenameObject()` but return the filename encoded to «utf-8».

Niezalecane od wersji 3.2: `PyModule_GetFilename()` raises `UnicodeEncodeError` on unencodable filenames, use `PyModule_GetFilenameObject()` instead.

Initializing C modules

Modules objects are usually created from extension modules (shared libraries which export an initialization function), or compiled-in modules (where the initialization function is added using `PyImport_AppendInittab()`). See building or extending-with-embedding for details.

The initialization function can either pass a module definition instance to `PyModule_Create()`, and return the resulting module object, or request „multi-phase initialization" by returning the definition struct itself.

type **PyModuleDef**

Część stabilnego ABI *(w tym wszystkie składniki).* The module definition struct, which holds all information needed to create a module object. There is usually only one statically initialized variable of this type for each module.

PyModuleDef_Base **m_base**

Always initialize this member to `PyModuleDef_HEAD_INIT`.

const char ***m_name**

Name for the new module.

const char ***m_doc**

Docstring for the module; usually a docstring variable created with `PyDoc_STRVAR` is used.

Py_ssize_t **m_size**

Module state may be kept in a per-module memory area that can be retrieved with `PyModule_GetState()`, rather than in static globals. This makes modules safe for use in multiple sub-interpreters.

This memory area is allocated based on *m_size* on module creation, and freed when the module object is deallocated, after the `m_free` function has been called, if present.

Setting `m_size` to `-1` means that the module does not support sub-interpreters, because it has global state.

Setting it to a non-negative value means that the module can be re-initialized and specifies the additional amount of memory it requires for its state. Non-negative `m_size` is required for multi-phase initialization.

See **PEP 3121** for more details.

PyMethodDef ***m_methods**

A pointer to a table of module-level functions, described by `PyMethodDef` values. Can be `NULL` if no functions are present.

PyModuleDef_Slot ***m_slots***

An array of slot definitions for multi-phase initialization, terminated by a `{0, NULL}` entry. When using single-phase initialization, *m_slots* must be `NULL`.

Zmienione w wersji 3.5: Prior to version 3.5, this member was always set to `NULL`, and was defined as:

inquiry **m_reload**

traverseproc **m_traverse**

A traversal function to call during GC traversal of the module object, or `NULL` if not needed.

This function is not called if the module state was requested but is not allocated yet. This is the case immediately after the module is created and before the module is executed (`Py_mod_exec` function). More precisely, this function is not called if `m_size` is greater than 0 and the module state (as returned by `PyModule_GetState()`) is `NULL`.

Zmienione w wersji 3.9: No longer called before the module state is allocated.

inquiry **m_clear**

A clear function to call during GC clearing of the module object, or `NULL` if not needed.

This function is not called if the module state was requested but is not allocated yet. This is the case immediately after the module is created and before the module is executed (`Py_mod_exec` function). More precisely, this function is not called if `m_size` is greater than 0 and the module state (as returned by `PyModule_GetState()`) is `NULL`.

Like `PyTypeObject.tp_clear`, this function is not *always* called before a module is deallocated. For example, when reference counting is enough to determine that an object is no longer used, the cyclic garbage collector is not involved and `m_free` is called directly.

Zmienione w wersji 3.9: No longer called before the module state is allocated.

freefunc **m_free**

A function to call during deallocation of the module object, or `NULL` if not needed.

This function is not called if the module state was requested but is not allocated yet. This is the case immediately after the module is created and before the module is executed (`Py_mod_exec` function). More precisely, this function is not called if `m_size` is greater than 0 and the module state (as returned by `PyModule_GetState()`) is `NULL`.

Zmienione w wersji 3.9: No longer called before the module state is allocated.

Single-phase initialization

The module initialization function may create and return the module object directly. This is referred to as „single--phase initialization", and uses one of the following two module creation functions:

PyObject ***PyModule_Create**(*PyModuleDef* *def)

Wartość zwracana: nowa referencja. Create a new module object, given the definition in *def*. This behaves like `PyModule_Create2()` with *module_api_version* set to `PYTHON_API_VERSION`.

PyObject ***PyModule_Create2**(*PyModuleDef* *def, int module_api_version)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Create a new module object, given the definition in *def*, assuming the API version *module_api_version*. If that version does not match the version of the running interpreter, a `RuntimeWarning` is emitted.

Return `NULL` with an exception set on error.

> **Informacja**
>
> Most uses of this function should be using `PyModule_Create()` instead; only use this if you are sure you need it.

Before it is returned from in the initialization function, the resulting module object is typically populated using functions like `PyModule_AddObjectRef()`.

Multi-phase initialization

An alternate way to specify extensions is to request „multi-phase initialization". Extension modules created this way behave more like Python modules: the initialization is split between the *creation phase*, when the module object is created, and the *execution phase*, when it is populated. The distinction is similar to the `__new__()` and `__init__()` methods of classes.

Unlike modules created using single-phase initialization, these modules are not singletons: if the *sys.modules* entry is removed and the module is re-imported, a new module object is created, and the old module is subject to normal garbage collection – as with Python modules. By default, multiple modules created from the same definition should be independent: changes to one should not affect the others. This means that all state should be specific to the module object (using e.g. using `PyModule_GetState()`), or its contents (such as the module's `__dict__` or individual classes created with `PyType_FromSpec()`).

All modules created using multi-phase initialization are expected to support *sub-interpreters*. Making sure multiple modules are independent is typically enough to achieve this.

To request multi-phase initialization, the initialization function (PyInit_modulename) returns a `PyModuleDef` instance with non-empty `m_slots`. Before it is returned, the `PyModuleDef` instance must be initialized with the following function:

PyObject ***PyModuleDef_Init**(*PyModuleDef* *def)

Wartość zwracana: pożyczona referencja. Część stabilnego ABI *od wersji 3.5.* Ensures a module definition is a properly initialized Python object that correctly reports its type and reference count.

Returns *def* cast to `PyObject*`, or `NULL` if an error occurred.

Added in version 3.5.

The *m_slots* member of the module definition must point to an array of `PyModuleDef_Slot` structures:

type **PyModuleDef_Slot**

int **slot**

A slot ID, chosen from the available values explained below.

void ***value**

Value of the slot, whose meaning depends on the slot ID.

Added in version 3.5.

The *m_slots* array must be terminated by a slot with id 0.

The available slot types are:

Py_mod_create

Specifies a function that is called to create the module object itself. The *value* pointer of this slot must point to a function of the signature:

PyObject ***create_module**(*PyObject* *spec, *PyModuleDef* *def)

The function receives a `ModuleSpec` instance, as defined in **PEP 451**, and the module definition. It should return a new module object, or set an error and return `NULL`.

This function should be kept minimal. In particular, it should not call arbitrary Python code, as trying to import the same module again may result in an infinite loop.

Multiple `Py_mod_create` slots may not be specified in one module definition.

If `Py_mod_create` is not specified, the import machinery will create a normal module object using `PyModule_New()`. The name is taken from *spec*, not the definition, to allow extension modules to dynamically adjust to their place in the module hierarchy and be imported under different names through symlinks, all while sharing a single module definition.

There is no requirement for the returned object to be an instance of `PyModule_Type`. Any type can be used, as long as it supports setting and getting import-related attributes. However, only `PyModule_Type` instances may be returned if the `PyModuleDef` has non-`NULL` `m_traverse`, `m_clear`, `m_free`; non-zero `m_size`; or slots other than `Py_mod_create`.

Py_mod_exec

Specifies a function that is called to *execute* the module. This is equivalent to executing the code of a Python module: typically, this function adds classes and constants to the module. The signature of the function is:

int **exec_module**(*PyObject* *module)

If multiple `Py_mod_exec` slots are specified, they are processed in the order they appear in the *m_slots* array.

Py_mod_multiple_interpreters

Specifies one of the following values:

Py_MOD_MULTIPLE_INTERPRETERS_NOT_SUPPORTED

The module does not support being imported in subinterpreters.

Py_MOD_MULTIPLE_INTERPRETERS_SUPPORTED

The module supports being imported in subinterpreters, but only when they share the main interpreter's GIL. (See isolating-extensions-howto.)

Py_MOD_PER_INTERPRETER_GIL_SUPPORTED

The module supports being imported in subinterpreters, even when they have their own GIL. (See isolating-extensions-howto.)

This slot determines whether or not importing this module in a subinterpreter will fail.

Multiple `Py_mod_multiple_interpreters` slots may not be specified in one module definition.

If `Py_mod_multiple_interpreters` is not specified, the import machinery defaults to `Py_MOD_MULTIPLE_INTERPRETERS_NOT_SUPPORTED`.

Added in version 3.12.

Py_mod_gil

Specifies one of the following values:

Py_MOD_GIL_USED

The module depends on the presence of the global interpreter lock (GIL), and may access global state without synchronization.

Py_MOD_GIL_NOT_USED

The module is safe to run without an active GIL.

This slot is ignored by Python builds not configured with `--disable-gil`. Otherwise, it determines whether or not importing this module will cause the GIL to be automatically enabled. See whatsnew313-free-threaded--cpython for more detail.

Multiple `Py_mod_gil` slots may not be specified in one module definition.

If `Py_mod_gil` is not specified, the import machinery defaults to `Py_MOD_GIL_USED`.

Added in version 3.13.

See **PEP 489** for more details on multi-phase initialization.

Low-level module creation functions

The following functions are called under the hood when using multi-phase initialization. They can be used directly, for example when creating module objects dynamically. Note that both `PyModule_FromDefAndSpec` and `PyModule_ExecDef` must be called to fully initialize a module.

PyObject ***PyModule_FromDefAndSpec*** (*PyModuleDef* *def, *PyObject* *spec)

Wartość zwracana: nowa referencja. Create a new module object, given the definition in *def* and the ModuleSpec *spec*. This behaves like `PyModule_FromDefAndSpec2()` with *module_api_version* set to `PYTHON_API_VERSION`.

Added in version 3.5.

PyObject ***PyModule_FromDefAndSpec2*** (*PyModuleDef* *def, *PyObject* *spec, int module_api_version)

Wartość zwracana: nowa referencja. Część stabilnego ABI *od wersji 3.7.* Create a new module object, given the definition in *def* and the ModuleSpec *spec*, assuming the API version *module_api_version*. If that version does not match the version of the running interpreter, a `RuntimeWarning` is emitted.

Return `NULL` with an exception set on error.

> **Informacja**
>
> Most uses of this function should be using `PyModule_FromDefAndSpec()` instead; only use this if you are sure you need it.

Added in version 3.5.

int ***PyModule_ExecDef*** (*PyObject* *module, *PyModuleDef* *def)

Część stabilnego ABI *od wersji 3.7.* Process any execution slots (`Py_mod_exec`) given in *def*.

Added in version 3.5.

int ***PyModule_SetDocString*** (*PyObject* *module, const char *docstring)

Część stabilnego ABI *od wersji 3.7.* Set the docstring for *module* to *docstring*. This function is called automatically when creating a module from `PyModuleDef`, using either `PyModule_Create` or `PyModule_FromDefAndSpec`.

Added in version 3.5.

int ***PyModule_AddFunctions*** (*PyObject* *module, *PyMethodDef* *functions)

Część stabilnego ABI *od wersji 3.7.* Add the functions from the `NULL` terminated *functions* array to *module*. Refer to the `PyMethodDef` documentation for details on individual entries (due to the lack of a shared module namespace, module level „functions" implemented in C typically receive the module as their first parameter, making them similar to instance methods on Python classes). This function is called automatically when creating a module from `PyModuleDef`, using either `PyModule_Create` or `PyModule_FromDefAndSpec`.

Added in version 3.5.

Support functions

The module initialization function (if using single phase initialization) or a function called from a module execution slot (if using multi-phase initialization), can use the following functions to help initialize the module state:

int ***PyModule_AddObjectRef*** (*PyObject* *module, const char *name, *PyObject* *value)

Część stabilnego ABI *od wersji 3.10.* Add an object to *module* as *name*. This is a convenience function which can be used from the module's initialization function.

On success, return `0`. On error, raise an exception and return `-1`.

Return `-1` if *value* is `NULL`. It must be called with an exception raised in this case.

Example usage:

```
static int
add_spam(PyObject *module, int value)
{
    PyObject *obj = PyLong_FromLong(value);
    if (obj == NULL) {
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
        return -1;
    }
    int res = PyModule_AddObjectRef(module, "spam", obj);
    Py_DECREF(obj);
    return res;
}
```

The example can also be written without checking explicitly if *obj* is `NULL`:

```
static int
add_spam(PyObject *module, int value)
{
    PyObject *obj = PyLong_FromLong(value);
    int res = PyModule_AddObjectRef(module, "spam", obj);
    Py_XDECREF(obj);
    return res;
}
```

Note that `Py_XDECREF()` should be used instead of `Py_DECREF()` in this case, since *obj* can be `NULL`.

The number of different *name* strings passed to this function should be kept small, usually by only using statically allocated strings as *name*. For names that aren't known at compile time, prefer calling `PyUnicode_FromString()` and `PyObject_SetAttr()` directly. For more details, see `PyUnicode_InternFromString()`, which may be used internally to create a key object.

Added in version 3.10.

int **PyModule_Add**(*PyObject* *module, const char *name, *PyObject* *value)

Część stabilnego ABI *od wersji 3.13*. Similar to `PyModule_AddObjectRef()`, but „steals" a reference to *value*. It can be called with a result of function that returns a new reference without bothering to check its result or even saving it to a variable.

Example usage:

```
if (PyModule_Add(module, "spam", PyBytes_FromString(value)) < 0) {
    goto error;
}
```

Added in version 3.13.

int **PyModule_AddObject**(*PyObject* *module, const char *name, *PyObject* *value)

Część stabilnego ABI. Similar to `PyModule_AddObjectRef()`, but steals a reference to *value* on success (if it returns `0`).

The new `PyModule_Add()` or `PyModule_AddObjectRef()` functions are recommended, since it is easy to introduce reference leaks by misusing the `PyModule_AddObject()` function.

Informacja

Unlike other functions that steal references, `PyModule_AddObject()` only releases the reference to *value* **on success**.

This means that its return value must be checked, and calling code must `Py_XDECREF()` *value* manually on error.

Example usage:

```
PyObject *obj = PyBytes_FromString(value);
if (PyModule_AddObject(module, "spam", obj) < 0) {
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
    // If 'obj' is not NULL and PyModule_AddObject() failed,
    // 'obj' strong reference must be deleted with Py_XDECREF().
    // If 'obj' is NULL, Py_XDECREF() does nothing.
    Py_XDECREF(obj);
    goto error;
}
// PyModule_AddObject() stole a reference to obj:
// Py_XDECREF(obj) is not needed here.
```

Niezalecane od wersji 3.13: `PyModule_AddObject()` is *soft deprecated*.

int **PyModule_AddIntConstant**(*PyObject* *module, const char *name, long value)

Część stabilnego ABI. Add an integer constant to *module* as *name*. This convenience function can be used from the module's initialization function. Return `-1` with an exception set on error, `0` on success.

This is a convenience function that calls `PyLong_FromLong()` and `PyModule_AddObjectRef()`; see their documentation for details.

int **PyModule_AddStringConstant**(*PyObject* *module, const char *name, const char *value)

Część stabilnego ABI. Add a string constant to *module* as *name*. This convenience function can be used from the module's initialization function. The string *value* must be `NULL`-terminated. Return `-1` with an exception set on error, `0` on success.

This is a convenience function that calls `PyUnicode_InternFromString()` and `PyModule_AddObjectRef()`; see their documentation for details.

PyModule_AddIntMacro(module, macro)

Add an int constant to *module*. The name and the value are taken from *macro*. For example `PyModule_AddIntMacro(module, AF_INET)` adds the int constant *AF_INET* with the value of *AF_INET* to *module*. Return `-1` with an exception set on error, `0` on success.

PyModule_AddStringMacro(module, macro)

Add a string constant to *module*.

int **PyModule_AddType**(*PyObject* *module, *PyTypeObject* *type)

Część stabilnego ABI *od wersji 3.10*. Add a type object to *module*. The type object is finalized by calling internally `PyType_Ready()`. The name of the type object is taken from the last component of `tp_name` after dot. Return `-1` with an exception set on error, `0` on success.

Added in version 3.9.

int **PyUnstable_Module_SetGIL**(*PyObject* *module, void *gil)

> To jest *niestabilne API*. Może ulec zmianie bez ostrzeżenia w wydaniach minor.

Indicate that *module* does or does not support running without the global interpreter lock (GIL), using one of the values from `Py_mod_gil`. It must be called during *module*'s initialization function. If this function is not called during module initialization, the import machinery assumes the module does not support running without the GIL. This function is only available in Python builds configured with `--disable-gil`. Return `-1` with an exception set on error, `0` on success.

Added in version 3.13.

Module lookup

Single-phase initialization creates singleton modules that can be looked up in the context of the current interpreter. This allows the module object to be retrieved later with only a reference to the module definition.

These functions will not work on modules created using multi-phase initialization, since multiple such modules can be created from a single definition.

PyObject ***PyState_FindModule**(*PyModuleDef* *def)

Wartość zwracana: pożyczona referencja. Część stabilnego ABI. Returns the module object that was created from *def* for the current interpreter. This method requires that the module object has been attached to the interpreter state with `PyState_AddModule()` beforehand. In case the corresponding module object is not found or has not been attached to the interpreter state yet, it returns `NULL`.

int **PyState_AddModule**(*PyObject* *module, *PyModuleDef* *def)

Część stabilnego ABI *od wersji 3.3.* Attaches the module object passed to the function to the interpreter state. This allows the module object to be accessible via `PyState_FindModule()`.

Only effective on modules created using single-phase initialization.

Python calls `PyState_AddModule` automatically after importing a module, so it is unnecessary (but harmless) to call it from module initialization code. An explicit call is needed only if the module's own init code subsequently calls `PyState_FindModule`. The function is mainly intended for implementing alternative import mechanisms (either by calling it directly, or by referring to its implementation for details of the required state updates).

The caller must hold the GIL.

Return `-1` with an exception set on error, `0` on success.

Added in version 3.3.

int **PyState_RemoveModule**(*PyModuleDef* *def)

Część stabilnego ABI *od wersji 3.3.* Removes the module object created from *def* from the interpreter state. Return `-1` with an exception set on error, `0` on success.

The caller must hold the GIL.

Added in version 3.3.

8.6.3 Iterator Objects

Python provides two general-purpose iterator objects. The first, a sequence iterator, works with an arbitrary sequence supporting the `__getitem__()` method. The second works with a callable object and a sentinel value, calling the callable for each item in the sequence, and ending the iteration when the sentinel value is returned.

PyTypeObject **PySeqIter_Type**

Część stabilnego ABI. Type object for iterator objects returned by `PySeqIter_New()` and the one-argument form of the `iter()` built-in function for built-in sequence types.

int **PySeqIter_Check**(*PyObject* *op)

Return true if the type of *op* is `PySeqIter_Type`. This function always succeeds.

PyObject ***PySeqIter_New**(*PyObject* *seq)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return an iterator that works with a general sequence object, *seq*. The iteration ends when the sequence raises `IndexError` for the subscripting operation.

PyTypeObject **PyCallIter_Type**

Część stabilnego ABI. Type object for iterator objects returned by `PyCallIter_New()` and the two-argument form of the `iter()` built-in function.

int **PyCallIter_Check**(*PyObject* *op)

Return true if the type of *op* is `PyCallIter_Type`. This function always succeeds.

PyObject ***PyCallIter_New*** (*PyObject* *callable, *PyObject* *sentinel)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return a new iterator. The first parameter, *callable*, can be any Python callable object that can be called with no parameters; each call to it should return the next item in the iteration. When *callable* returns a value equal to *sentinel*, the iteration will be terminated.

8.6.4 Descriptor Objects

„Descriptors" are objects that describe some attribute of an object. They are found in the dictionary of type objects.

PyTypeObject **PyProperty_Type**

Część stabilnego ABI. The type object for the built-in descriptor types.

PyObject ***PyDescr_NewGetSet*** (*PyTypeObject* *type, struct *PyGetSetDef* *getset)

Wartość zwracana: nowa referencja. Część stabilnego ABI.

PyObject ***PyDescr_NewMember*** (*PyTypeObject* *type, struct *PyMemberDef* *meth)

Wartość zwracana: nowa referencja. Część stabilnego ABI.

PyObject ***PyDescr_NewMethod*** (*PyTypeObject* *type, struct *PyMethodDef* *meth)

Wartość zwracana: nowa referencja. Część stabilnego ABI.

PyObject ***PyDescr_NewWrapper*** (*PyTypeObject* *type, struct wrapperbase *wrapper, void *wrapped)

Wartość zwracana: nowa referencja.

PyObject ***PyDescr_NewClassMethod*** (*PyTypeObject* *type, *PyMethodDef* *method)

Wartość zwracana: nowa referencja. Część stabilnego ABI.

int ***PyDescr_IsData*** (*PyObject* *descr)

Return non-zero if the descriptor objects *descr* describes a data attribute, or `0` if it describes a method. *descr* must be a descriptor object; there is no error checking.

PyObject ***PyWrapper_New*** (*PyObject**, *PyObject**)

Wartość zwracana: nowa referencja. Część stabilnego ABI.

8.6.5 Slice Objects

PyTypeObject **PySlice_Type**

Część stabilnego ABI. The type object for slice objects. This is the same as `slice` in the Python layer.

int ***PySlice_Check*** (*PyObject* *ob)

Return true if *ob* is a slice object; *ob* must not be `NULL`. This function always succeeds.

PyObject ***PySlice_New*** (*PyObject* *start, *PyObject* *stop, *PyObject* *step)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return a new slice object with the given values. The *start*, *stop*, and *step* parameters are used as the values of the slice object attributes of the same names. Any of the values may be `NULL`, in which case the `None` will be used for the corresponding attribute.

Return `NULL` with an exception set if the new object could not be allocated.

int ***PySlice_GetIndices*** (*PyObject* *slice, *Py_ssize_t* length, *Py_ssize_t* *start, *Py_ssize_t* *stop, *Py_ssize_t* *step)

Część stabilnego ABI. Retrieve the start, stop and step indices from the slice object *slice*, assuming a sequence of length *length*. Treats indices greater than *length* as errors.

Returns `0` on success and `-1` on error with no exception set (unless one of the indices was not `None` and failed to be converted to an integer, in which case `-1` is returned with an exception set).

You probably do not want to use this function.

Zmienione w wersji 3.2: The parameter type for the *slice* parameter was `PySliceObject*` before.

int **PySlice_GetIndicesEx**(*PyObject* *slice, *Py_ssize_t* length, *Py_ssize_t* *start, *Py_ssize_t* *stop, *Py_ssize_t* *step, *Py_ssize_t* *slicelength)

Część stabilnego ABI. Usable replacement for `PySlice_GetIndices()`. Retrieve the start, stop, and step indices from the slice object *slice* assuming a sequence of length *length*, and store the length of the slice in *slicelength*. Out of bounds indices are clipped in a manner consistent with the handling of normal slices.

Return `0` on success and `-1` on error with an exception set.

Informacja

This function is considered not safe for resizable sequences. Its invocation should be replaced by a combination of `PySlice_Unpack()` and `PySlice_AdjustIndices()` where

```
if (PySlice_GetIndicesEx(slice, length, &start, &stop, &step, &slicelength)
↪< 0) {
    // return error
}
```

is replaced by

```
if (PySlice_Unpack(slice, &start, &stop, &step) < 0) {
    // return error
}
slicelength = PySlice_AdjustIndices(length, &start, &stop, step);
```

Zmienione w wersji 3.2: The parameter type for the *slice* parameter was `PySliceObject*` before.

Zmienione w wersji 3.6.1: If `Py_LIMITED_API` is not set or set to the value between `0x03050400` and `0x03060000` (not including) or `0x03060100` or higher `PySlice_GetIndicesEx()` is implemented as a macro using `PySlice_Unpack()` and `PySlice_AdjustIndices()`. Arguments *start*, *stop* and *step* are evaluated more than once.

Niezalecane od wersji 3.6.1: If `Py_LIMITED_API` is set to the value less than `0x03050400` or between `0x03060000` and `0x03060100` (not including) `PySlice_GetIndicesEx()` is a deprecated function.

int **PySlice_Unpack**(*PyObject* *slice, *Py_ssize_t* *start, *Py_ssize_t* *stop, *Py_ssize_t* *step)

Część stabilnego ABI *od wersji 3.7.* Extract the start, stop and step data members from a slice object as C integers. Silently reduce values larger than `PY_SSIZE_T_MAX` to `PY_SSIZE_T_MAX`, silently boost the start and stop values less than `PY_SSIZE_T_MIN` to `PY_SSIZE_T_MIN`, and silently boost the step values less than `-PY_SSIZE_T_MAX` to `-PY_SSIZE_T_MAX`.

Return `-1` with an exception set on error, `0` on success.

Added in version 3.6.1.

Py_ssize_t **PySlice_AdjustIndices**(*Py_ssize_t* length, *Py_ssize_t* *start, *Py_ssize_t* *stop, *Py_ssize_t* step)

Część stabilnego ABI *od wersji 3.7.* Adjust start/end slice indices assuming a sequence of the specified length. Out of bounds indices are clipped in a manner consistent with the handling of normal slices.

Return the length of the slice. Always successful. Doesn't call Python code.

Added in version 3.6.1.

Ellipsis Object

PyObject ***Py_Ellipsis**

The Python `Ellipsis` object. This object has no methods. Like `Py_None`, it is an *immortal* singleton object.

Zmienione w wersji 3.12: `Py_Ellipsis` is immortal.

8.6.6 MemoryView objects

A `memoryview` object exposes the C level *buffer interface* as a Python object which can then be passed around like any other object.

PyObject ***PyMemoryView_FromObject**(*PyObject* *obj)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Create a memoryview object from an object that provides the buffer interface. If *obj* supports writable buffer exports, the memoryview object will be read/write, otherwise it may be either read-only or read/write at the discretion of the exporter.

PyBUF_READ

Flag to request a readonly buffer.

PyBUF_WRITE

Flag to request a writable buffer.

PyObject ***PyMemoryView_FromMemory**(char *mem, *Py_ssize_t* size, int flags)

Wartość zwracana: nowa referencja. Część stabilnego ABI *od wersji 3.7.* Create a memoryview object using *mem* as the underlying buffer. *flags* can be one of `PyBUF_READ` or `PyBUF_WRITE`.

Added in version 3.3.

PyObject ***PyMemoryView_FromBuffer**(const *Py_buffer* *view)

Wartość zwracana: nowa referencja. Część stabilnego ABI *od wersji 3.11.* Create a memoryview object wrapping the given buffer structure *view*. For simple byte buffers, `PyMemoryView_FromMemory()` is the preferred function.

PyObject ***PyMemoryView_GetContiguous**(*PyObject* *obj, int buffertype, char order)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Create a memoryview object to a *contiguous* chunk of memory (in either «C» or «F'ortran *order*) from an object that defines the buffer interface. If memory is contiguous, the memoryview object points to the original memory. Otherwise, a copy is made and the memoryview points to a new bytes object.

buffertype can be one of `PyBUF_READ` or `PyBUF_WRITE`.

int **PyMemoryView_Check**(*PyObject* *obj)

Return true if the object *obj* is a memoryview object. It is not currently allowed to create subclasses of `memoryview`. This function always succeeds.

Py_buffer ***PyMemoryView_GET_BUFFER**(*PyObject* *mview)

Return a pointer to the memoryview's private copy of the exporter's buffer. *mview* **must** be a memoryview instance; this macro doesn't check its type, you must do it yourself or you will risk crashes.

PyObject ***PyMemoryView_GET_BASE**(*PyObject* *mview)

Return either a pointer to the exporting object that the memoryview is based on or `NULL` if the memoryview has been created by one of the functions `PyMemoryView_FromMemory()` or `PyMemoryView_FromBuffer()`. *mview* **must** be a memoryview instance.

8.6.7 Weak Reference Objects

Python supports *weak references* as first-class objects. There are two specific object types which directly implement weak references. The first is a simple reference object, and the second acts as a proxy for the original object as much as it can.

int **PyWeakref_Check**(*PyObject* *ob)

Return non-zero if *ob* is either a reference or proxy object. This function always succeeds.

int **PyWeakref_CheckRef**(*PyObject* *ob)

Return non-zero if *ob* is a reference object. This function always succeeds.

int **PyWeakref_CheckProxy**(*PyObject* *ob)

Return non-zero if *ob* is a proxy object. This function always succeeds.

PyObject ***PyWeakref_NewRef*** (*PyObject* *ob, *PyObject* *callback)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return a weak reference object for the object *ob*. This will always return a new reference, but is not guaranteed to create a new object; an existing reference object may be returned. The second parameter, *callback*, can be a callable object that receives notification when *ob* is garbage collected; it should accept a single parameter, which will be the weak reference object itself. *callback* may also be `None` or `NULL`. If *ob* is not a weakly referenceable object, or if *callback* is not callable, `None`, or `NULL`, this will return `NULL` and raise `TypeError`.

PyObject ***PyWeakref_NewProxy*** (*PyObject* *ob, *PyObject* *callback)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Return a weak reference proxy object for the object *ob*. This will always return a new reference, but is not guaranteed to create a new object; an existing proxy object may be returned. The second parameter, *callback*, can be a callable object that receives notification when *ob* is garbage collected; it should accept a single parameter, which will be the weak reference object itself. *callback* may also be `None` or `NULL`. If *ob* is not a weakly referenceable object, or if *callback* is not callable, `None`, or `NULL`, this will return `NULL` and raise `TypeError`.

int ***PyWeakref_GetRef*** (*PyObject* *ref, *PyObject* **pobj)

Część stabilnego ABI *od wersji 3.13.* Get a *strong reference* to the referenced object from a weak reference, *ref*, into **pobj*.

- On success, set **pobj* to a new *strong reference* to the referenced object and return 1.
- If the reference is dead, set **pobj* to `NULL` and return 0.
- On error, raise an exception and return -1.

Added in version 3.13.

PyObject ***PyWeakref_GetObject*** (*PyObject* *ref)

Wartość zwracana: pożyczona referencja. Część stabilnego ABI. Return a *borrowed reference* to the referenced object from a weak reference, *ref*. If the referent is no longer live, returns `Py_None`.

> **Informacja**
>
> This function returns a *borrowed reference* to the referenced object. This means that you should always call `Py_INCREF()` on the object except when it cannot be destroyed before the last usage of the borrowed reference.

Deprecated since version 3.13, will be removed in version 3.15: Use `PyWeakref_GetRef()` instead.

PyObject ***PyWeakref_GET_OBJECT*** (*PyObject* *ref)

Wartość zwracana: pożyczona referencja. Similar to `PyWeakref_GetObject()`, but does no error checking.

Deprecated since version 3.13, will be removed in version 3.15: Use `PyWeakref_GetRef()` instead.

void ***PyObject_ClearWeakRefs*** (*PyObject* *object)

Część stabilnego ABI. This function is called by the `tp_dealloc` handler to clear weak references.

This iterates through the weak references for *object* and calls callbacks for those references which have one. It returns when all callbacks have been attempted.

void ***PyUnstable_Object_ClearWeakRefsNoCallbacks*** (*PyObject* *object)

> To jest *niestabilne API*. Może ulec zmianie bez ostrzeżenia w wydaniach minor.

Clears the weakrefs for *object* without calling the callbacks.

This function is called by the `tp_dealloc` handler for types with finalizers (i.e., `__del__()`). The handler for those objects first calls `PyObject_ClearWeakRefs()` to clear weakrefs and call their callbacks, then the finalizer, and finally this function to clear any weakrefs that may have been created by the finalizer.

In most circumstances, it's more appropriate to use `PyObject_ClearWeakRefs()` to clear weakrefs instead of this function.

Added in version 3.13.

8.6.8 Capsules

Refer to using-capsules for more information on using these objects.

Added in version 3.1.

type **PyCapsule**

This subtype of `PyObject` represents an opaque value, useful for C extension modules who need to pass an opaque value (as a `void*` pointer) through Python code to other C code. It is often used to make a C function pointer defined in one module available to other modules, so the regular import mechanism can be used to access C APIs defined in dynamically loaded modules.

type **PyCapsule_Destructor**

Część stabilnego ABI. The type of a destructor callback for a capsule. Defined as:

```
typedef void (*PyCapsule_Destructor)(PyObject *);
```

See `PyCapsule_New()` for the semantics of PyCapsule_Destructor callbacks.

int **PyCapsule_CheckExact**(*PyObject* *p)

Return true if its argument is a `PyCapsule`. This function always succeeds.

PyObject ***PyCapsule_New**(void *pointer, const char *name, *PyCapsule_Destructor* destructor)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Create a `PyCapsule` encapsulating the *pointer*. The *pointer* argument may not be `NULL`.

On failure, set an exception and return `NULL`.

The *name* string may either be `NULL` or a pointer to a valid C string. If non-`NULL`, this string must outlive the capsule. (Though it is permitted to free it inside the *destructor*.)

If the *destructor* argument is not `NULL`, it will be called with the capsule as its argument when it is destroyed.

If this capsule will be stored as an attribute of a module, the *name* should be specified as `modulename.attributename`. This will enable other modules to import the capsule using `PyCapsule_Import()`.

void ***PyCapsule_GetPointer**(*PyObject* *capsule, const char *name)

Część stabilnego ABI. Retrieve the *pointer* stored in the capsule. On failure, set an exception and return `NULL`.

The *name* parameter must compare exactly to the name stored in the capsule. If the name stored in the capsule is `NULL`, the *name* passed in must also be `NULL`. Python uses the C function `strcmp()` to compare capsule names.

PyCapsule_Destructor **PyCapsule_GetDestructor**(*PyObject* *capsule)

Część stabilnego ABI. Return the current destructor stored in the capsule. On failure, set an exception and return `NULL`.

It is legal for a capsule to have a `NULL` destructor. This makes a `NULL` return code somewhat ambiguous; use `PyCapsule_IsValid()` or `PyErr_Occurred()` to disambiguate.

void ***PyCapsule_GetContext**(*PyObject* *capsule)

Część stabilnego ABI. Return the current context stored in the capsule. On failure, set an exception and return `NULL`.

It is legal for a capsule to have a `NULL` context. This makes a `NULL` return code somewhat ambiguous; use `PyCapsule_IsValid()` or `PyErr_Occurred()` to disambiguate.

const char ***PyCapsule_GetName**(*PyObject* *capsule)

Część stabilnego ABI. Return the current name stored in the capsule. On failure, set an exception and return `NULL`.

It is legal for a capsule to have a `NULL` name. This makes a `NULL` return code somewhat ambiguous; use `PyCapsule_IsValid()` or `PyErr_Occurred()` to disambiguate.

void ***PyCapsule_Import**(const char *name, int no_block)

Część stabilnego ABI. Import a pointer to a C object from a capsule attribute in a module. The *name* parameter should specify the full name to the attribute, as in `module.attribute`. The *name* stored in the capsule must match this string exactly.

Return the capsule's internal *pointer* on success. On failure, set an exception and return `NULL`.

Zmienione w wersji 3.3: *no_block* has no effect anymore.

int **PyCapsule_IsValid**(*PyObject* *capsule, const char *name)

Część stabilnego ABI. Determines whether or not *capsule* is a valid capsule. A valid capsule is non-`NULL`, passes `PyCapsule_CheckExact()`, has a non-`NULL` pointer stored in it, and its internal name matches the *name* parameter. (See `PyCapsule_GetPointer()` for information on how capsule names are compared.)

In other words, if `PyCapsule_IsValid()` returns a true value, calls to any of the accessors (any function starting with `PyCapsule_Get`) are guaranteed to succeed.

Return a nonzero value if the object is valid and matches the name passed in. Return `0` otherwise. This function will not fail.

int **PyCapsule_SetContext**(*PyObject* *capsule, void *context)

Część stabilnego ABI. Set the context pointer inside *capsule* to *context*.

Return `0` on success. Return nonzero and set an exception on failure.

int **PyCapsule_SetDestructor**(*PyObject* *capsule, *PyCapsule_Destructor* destructor)

Część stabilnego ABI. Set the destructor inside *capsule* to *destructor*.

Return `0` on success. Return nonzero and set an exception on failure.

int **PyCapsule_SetName**(*PyObject* *capsule, const char *name)

Część stabilnego ABI. Set the name inside *capsule* to *name*. If non-`NULL`, the name must outlive the capsule. If the previous *name* stored in the capsule was not `NULL`, no attempt is made to free it.

Return `0` on success. Return nonzero and set an exception on failure.

int **PyCapsule_SetPointer**(*PyObject* *capsule, void *pointer)

Część stabilnego ABI. Set the void pointer inside *capsule* to *pointer*. The pointer may not be `NULL`.

Return `0` on success. Return nonzero and set an exception on failure.

8.6.9 Frame Objects

type **PyFrameObject**

Część ograniczonego API *(jako nieprzezroczysta struktura)*. The C structure of the objects used to describe frame objects.

There are no public members in this structure.

Zmienione w wersji 3.11: The members of this structure were removed from the public C API. Refer to the What's New entry for details.

The `PyEval_GetFrame()` and `PyThreadState_GetFrame()` functions can be used to get a frame object.

See also *Reflection*.

PyTypeObject **PyFrame_Type**

The type of frame objects. It is the same object as `types.FrameType` in the Python layer.

Zmienione w wersji 3.11: Previously, this type was only available after including `<frameobject.h>`.

int **PyFrame_Check**(*PyObject* *obj)

Return non-zero if *obj* is a frame object.

Zmienione w wersji 3.11: Previously, this function was only available after including `<frameobject.h>`.

PyFrameObject ***PyFrame_GetBack**(*PyFrameObject* *frame)

Wartość zwracana: nowa referencja. Get the *frame* next outer frame.

Return a *strong reference*, or `NULL` if *frame* has no outer frame.

Added in version 3.9.

PyObject ***PyFrame_GetBuiltins**(*PyFrameObject* *frame)

Wartość zwracana: nowa referencja. Get the *frame*'s `f_builtins` attribute.

Return a *strong reference*. The result cannot be `NULL`.

Added in version 3.11.

PyCodeObject ***PyFrame_GetCode**(*PyFrameObject* *frame)

Wartość zwracana: nowa referencja. Część stabilnego ABI *od wersji 3.10.* Get the *frame* code.

Return a *strong reference*.

The result (frame code) cannot be `NULL`.

Added in version 3.9.

PyObject ***PyFrame_GetGenerator**(*PyFrameObject* *frame)

Wartość zwracana: nowa referencja. Get the generator, coroutine, or async generator that owns this frame, or `NULL` if this frame is not owned by a generator. Does not raise an exception, even if the return value is `NULL`.

Return a *strong reference*, or `NULL`.

Added in version 3.11.

PyObject ***PyFrame_GetGlobals**(*PyFrameObject* *frame)

Wartość zwracana: nowa referencja. Get the *frame*'s `f_globals` attribute.

Return a *strong reference*. The result cannot be `NULL`.

Added in version 3.11.

int **PyFrame_GetLasti**(*PyFrameObject* *frame)

Get the *frame*'s `f_lasti` attribute.

Returns -1 if `frame.f_lasti` is `None`.

Added in version 3.11.

PyObject ***PyFrame_GetVar**(*PyFrameObject* *frame, *PyObject* *name)

Wartość zwracana: nowa referencja. Get the variable *name* of *frame*.

- Return a *strong reference* to the variable value on success.
- Raise `NameError` and return `NULL` if the variable does not exist.
- Raise an exception and return `NULL` on error.

name type must be a `str`.

Added in version 3.12.

PyObject ***PyFrame_GetVarString*** (*PyFrameObject* *frame, const char *name)

Wartość zwracana: nowa referencja. Similar to `PyFrame_GetVar()`, but the variable name is a C string encoded in UTF-8.

Added in version 3.12.

PyObject ***PyFrame_GetLocals*** (*PyFrameObject* *frame)

Wartość zwracana: nowa referencja. Get the *frame*'s `f_locals` attribute. If the frame refers to an *optimized scope*, this returns a write-through proxy object that allows modifying the locals. In all other cases (classes, modules, `exec()`, `eval()`) it returns the mapping representing the frame locals directly (as described for `locals()`).

Return a *strong reference*.

Added in version 3.11.

Zmienione w wersji 3.13: As part of **PEP 667**, return a proxy object for optimized scopes.

int ***PyFrame_GetLineNumber*** (*PyFrameObject* *frame)

Część stabilnego ABI *od wersji 3.10.* Return the line number that *frame* is currently executing.

Internal Frames

Unless using **PEP 523**, you will not need this.

struct ***_PyInterpreterFrame***

The interpreter's internal frame representation.

Added in version 3.11.

PyObject ***PyUnstable_InterpreterFrame_GetCode*** (struct *_PyInterpreterFrame* *frame);

> To jest *niestabilne API*. Może ulec zmianie bez ostrzeżenia w wydaniach minor.

Return a *strong reference* to the code object for the frame.

Added in version 3.12.

int ***PyUnstable_InterpreterFrame_GetLasti*** (struct *_PyInterpreterFrame* *frame);

> To jest *niestabilne API*. Może ulec zmianie bez ostrzeżenia w wydaniach minor.

Return the byte offset into the last executed instruction.

Added in version 3.12.

int ***PyUnstable_InterpreterFrame_GetLine*** (struct *_PyInterpreterFrame* *frame);

> To jest *niestabilne API*. Może ulec zmianie bez ostrzeżenia w wydaniach minor.

Return the currently executing line number, or -1 if there is no line number.

Added in version 3.12.

8.6.10 Generator Objects

Generator objects are what Python uses to implement generator iterators. They are normally created by iterating over a function that yields values, rather than explicitly calling `PyGen_New()` or `PyGen_NewWithQualName()`.

type **PyGenObject**

The C structure used for generator objects.

PyTypeObject **PyGen_Type**

The type object corresponding to generator objects.

int **PyGen_Check**(*PyObject* *ob)

Return true if *ob* is a generator object; *ob* must not be `NULL`. This function always succeeds.

int **PyGen_CheckExact**(*PyObject* *ob)

Return true if *ob*'s type is `PyGen_Type`; *ob* must not be `NULL`. This function always succeeds.

PyObject ***PyGen_New**(*PyFrameObject* *frame)

Wartość zwracana: nowa referencja. Create and return a new generator object based on the *frame* object. A reference to *frame* is stolen by this function. The argument must not be `NULL`.

PyObject ***PyGen_NewWithQualName**(*PyFrameObject* *frame, *PyObject* *name, *PyObject* *qualname)

Wartość zwracana: nowa referencja. Create and return a new generator object based on the *frame* object, with `__name__` and `__qualname__` set to *name* and *qualname*. A reference to *frame* is stolen by this function. The *frame* argument must not be `NULL`.

8.6.11 Coroutine Objects

Added in version 3.5.

Coroutine objects are what functions declared with an `async` keyword return.

type **PyCoroObject**

The C structure used for coroutine objects.

PyTypeObject **PyCoro_Type**

The type object corresponding to coroutine objects.

int **PyCoro_CheckExact**(*PyObject* *ob)

Return true if *ob*'s type is `PyCoro_Type`; *ob* must not be `NULL`. This function always succeeds.

PyObject ***PyCoro_New**(*PyFrameObject* *frame, *PyObject* *name, *PyObject* *qualname)

Wartość zwracana: nowa referencja. Create and return a new coroutine object based on the *frame* object, with `__name__` and `__qualname__` set to *name* and *qualname*. A reference to *frame* is stolen by this function. The *frame* argument must not be `NULL`.

8.6.12 Context Variables Objects

Added in version 3.7.

Zmienione w wersji 3.7.1:

Informacja

In Python 3.7.1 the signatures of all context variables C APIs were **changed** to use `PyObject` pointers instead of `PyContext`, `PyContextVar`, and `PyContextToken`, e.g.:

```
// in 3.7.0:
PyContext *PyContext_New(void);

// in 3.7.1+:
PyObject *PyContext_New(void);
```

See bpo-34762 for more details.

This section details the public C API for the `contextvars` module.

type **PyContext**

The C structure used to represent a `contextvars.Context` object.

type **PyContextVar**

The C structure used to represent a `contextvars.ContextVar` object.

type **PyContextToken**

The C structure used to represent a `contextvars.Token` object.

PyTypeObject **PyContext_Type**

The type object representing the *context* type.

PyTypeObject **PyContextVar_Type**

The type object representing the *context variable* type.

PyTypeObject **PyContextToken_Type**

The type object representing the *context variable token* type.

Type-check macros:

int **PyContext_CheckExact**(*PyObject* *o)

Return true if *o* is of type `PyContext_Type`. *o* must not be `NULL`. This function always succeeds.

int **PyContextVar_CheckExact**(*PyObject* *o)

Return true if *o* is of type `PyContextVar_Type`. *o* must not be `NULL`. This function always succeeds.

int **PyContextToken_CheckExact**(*PyObject* *o)

Return true if *o* is of type `PyContextToken_Type`. *o* must not be `NULL`. This function always succeeds.

Context object management functions:

PyObject ***PyContext_New**(void)

Wartość zwracana: nowa referencja. Create a new empty context object. Returns `NULL` if an error has occurred.

PyObject ***PyContext_Copy**(*PyObject* *ctx)

Wartość zwracana: nowa referencja. Create a shallow copy of the passed *ctx* context object. Returns `NULL` if an error has occurred.

PyObject ***PyContext_CopyCurrent**(void)

Wartość zwracana: nowa referencja. Create a shallow copy of the current thread context. Returns `NULL` if an error has occurred.

int **PyContext_Enter**(*PyObject* *ctx)

Set *ctx* as the current context for the current thread. Returns `0` on success, and `-1` on error.

int **PyContext_Exit**(*PyObject* *ctx)

Deactivate the *ctx* context and restore the previous context as the current context for the current thread. Returns `0` on success, and `-1` on error.

Context variable functions:

PyObject ***PyContextVar_New**(const char *name, *PyObject* *def)

Wartość zwracana: nowa referencja. Create a new `ContextVar` object. The *name* parameter is used for introspection and debug purposes. The *def* parameter specifies a default value for the context variable, or `NULL` for no default. If an error has occurred, this function returns `NULL`.

int **PyContextVar_Get**(*PyObject* *var, *PyObject* *default_value, *PyObject* **value)

Get the value of a context variable. Returns `-1` if an error has occurred during lookup, and `0` if no error occurred, whether or not a value was found.

If the context variable was found, *value* will be a pointer to it. If the context variable was *not* found, *value* will point to:

- *default_value*, if not `NULL`;
- the default value of *var*, if not `NULL`;
- `NULL`

Except for `NULL`, the function returns a new reference.

PyObject ***PyContextVar_Set**(*PyObject* *var, *PyObject* *value)

Wartość zwracana: nowa referencja. Set the value of *var* to *value* in the current context. Returns a new token object for this change, or `NULL` if an error has occurred.

int **PyContextVar_Reset**(*PyObject* *var, *PyObject* *token)

Reset the state of the *var* context variable to that it was in before `PyContextVar_Set()` that returned the *token* was called. This function returns `0` on success and `-1` on error.

8.6.13 DateTime Objects

Various date and time objects are supplied by the `datetime` module. Before using any of these functions, the header file `datetime.h` must be included in your source (note that this is not included by `Python.h`), and the macro `PyDateTime_IMPORT` must be invoked, usually as part of the module initialisation function. The macro puts a pointer to a C structure into a static variable, `PyDateTimeAPI`, that is used by the following macros.

type **PyDateTime_Date**

This subtype of `PyObject` represents a Python date object.

type **PyDateTime_DateTime**

This subtype of `PyObject` represents a Python datetime object.

type **PyDateTime_Time**

This subtype of `PyObject` represents a Python time object.

type **PyDateTime_Delta**

This subtype of `PyObject` represents the difference between two datetime values.

PyTypeObject **PyDateTime_DateType**

This instance of `PyTypeObject` represents the Python date type; it is the same object as `datetime.date` in the Python layer.

PyTypeObject **PyDateTime_DateTimeType**

This instance of `PyTypeObject` represents the Python datetime type; it is the same object as `datetime.datetime` in the Python layer.

PyTypeObject **PyDateTime_TimeType**

This instance of `PyTypeObject` represents the Python time type; it is the same object as `datetime.time` in the Python layer.

PyTypeObject **PyDateTime_DeltaType**

This instance of `PyTypeObject` represents Python type for the difference between two datetime values; it is the same object as `datetime.timedelta` in the Python layer.

PyTypeObject **PyDateTime_TZInfoType**

This instance of `PyTypeObject` represents the Python time zone info type; it is the same object as `datetime.tzinfo` in the Python layer.

Macro for access to the UTC singleton:

PyObject ***PyDateTime_TimeZone_UTC**

Returns the time zone singleton representing UTC, the same object as `datetime.timezone.utc`.

Added in version 3.7.

Type-check macros:

int **PyDate_Check**(*PyObject* *ob)

Return true if *ob* is of type `PyDateTime_DateType` or a subtype of `PyDateTime_DateType`. *ob* must not be `NULL`. This function always succeeds.

int **PyDate_CheckExact**(*PyObject* *ob)

Return true if *ob* is of type `PyDateTime_DateType`. *ob* must not be `NULL`. This function always succeeds.

int **PyDateTime_Check**(*PyObject* *ob)

Return true if *ob* is of type `PyDateTime_DateTimeType` or a subtype of `PyDateTime_DateTimeType`. *ob* must not be `NULL`. This function always succeeds.

int **PyDateTime_CheckExact**(*PyObject* *ob)

Return true if *ob* is of type `PyDateTime_DateTimeType`. *ob* must not be `NULL`. This function always succeeds.

int **PyTime_Check**(*PyObject* *ob)

Return true if *ob* is of type `PyDateTime_TimeType` or a subtype of `PyDateTime_TimeType`. *ob* must not be `NULL`. This function always succeeds.

int **PyTime_CheckExact**(*PyObject* *ob)

Return true if *ob* is of type `PyDateTime_TimeType`. *ob* must not be `NULL`. This function always succeeds.

int **PyDelta_Check**(*PyObject* *ob)

Return true if *ob* is of type `PyDateTime_DeltaType` or a subtype of `PyDateTime_DeltaType`. *ob* must not be `NULL`. This function always succeeds.

int **PyDelta_CheckExact**(*PyObject* *ob)

Return true if *ob* is of type `PyDateTime_DeltaType`. *ob* must not be `NULL`. This function always succeeds.

int **PyTZInfo_Check**(*PyObject* *ob)

Return true if *ob* is of type `PyDateTime_TZInfoType` or a subtype of `PyDateTime_TZInfoType`. *ob* must not be `NULL`. This function always succeeds.

int **PyTZInfo_CheckExact**(*PyObject* *ob)

Return true if *ob* is of type `PyDateTime_TZInfoType`. *ob* must not be `NULL`. This function always succeeds.

Macros to create objects:

PyObject ***PyDate_FromDate**(int year, int month, int day)

Wartość zwracana: nowa referencja. Return a `datetime.date` object with the specified year, month and day.

PyObject ***PyDateTime_FromDateAndTime**(int year, int month, int day, int hour, int minute, int second, int usecond)

Wartość zwracana: nowa referencja. Return a `datetime.datetime` object with the specified year, month, day, hour, minute, second and microsecond.

PyObject ***PyDateTime_FromDateAndTimeAndFold**(int year, int month, int day, int hour, int minute, int second, int usecond, int fold)

Wartość zwracana: nowa referencja. Return a `datetime.datetime` object with the specified year, month, day, hour, minute, second, microsecond and fold.

Added in version 3.6.

PyObject ***PyTime_FromTime**(int hour, int minute, int second, int usecond)

Wartość zwracana: nowa referencja. Return a `datetime.time` object with the specified hour, minute, second and microsecond.

PyObject ***PyTime_FromTimeAndFold**(int hour, int minute, int second, int usecond, int fold)

Wartość zwracana: nowa referencja. Return a `datetime.time` object with the specified hour, minute, second, microsecond and fold.

Added in version 3.6.

PyObject ***PyDelta_FromDSU**(int days, int seconds, int useconds)

Wartość zwracana: nowa referencja. Return a `datetime.timedelta` object representing the given number of days, seconds and microseconds. Normalization is performed so that the resulting number of microseconds and seconds lie in the ranges documented for `datetime.timedelta` objects.

PyObject ***PyTimeZone_FromOffset**(*PyObject* *offset)

Wartość zwracana: nowa referencja. Return a `datetime.timezone` object with an unnamed fixed offset represented by the *offset* argument.

Added in version 3.7.

PyObject ***PyTimeZone_FromOffsetAndName**(*PyObject* *offset, *PyObject* *name)

Wartość zwracana: nowa referencja. Return a `datetime.timezone` object with a fixed offset represented by the *offset* argument and with tzname *name*.

Added in version 3.7.

Macros to extract fields from date objects. The argument must be an instance of `PyDateTime_Date`, including subclasses (such as `PyDateTime_DateTime`). The argument must not be `NULL`, and the type is not checked:

int **PyDateTime_GET_YEAR**(*PyDateTime_Date* *o)

Return the year, as a positive int.

int **PyDateTime_GET_MONTH**(*PyDateTime_Date* *o)

Return the month, as an int from 1 through 12.

int **PyDateTime_GET_DAY**(*PyDateTime_Date* *o)

Return the day, as an int from 1 through 31.

Macros to extract fields from datetime objects. The argument must be an instance of `PyDateTime_DateTime`, including subclasses. The argument must not be `NULL`, and the type is not checked:

int **PyDateTime_DATE_GET_HOUR**(*PyDateTime_DateTime* *o)

Return the hour, as an int from 0 through 23.

int **PyDateTime_DATE_GET_MINUTE**(*PyDateTime_DateTime* *o)

Return the minute, as an int from 0 through 59.

int **PyDateTime_DATE_GET_SECOND**(*PyDateTime_DateTime* *o)

Return the second, as an int from 0 through 59.

int **PyDateTime_DATE_GET_MICROSECOND**(*PyDateTime_DateTime* *o)

Return the microsecond, as an int from 0 through 999999.

int **PyDateTime_DATE_GET_FOLD**(*PyDateTime_DateTime* *o)

Return the fold, as an int from 0 through 1.

Added in version 3.6.

PyObject ***PyDateTime_DATE_GET_TZINFO**(*PyDateTime_DateTime* *o)

Return the tzinfo (which may be `None`).

Added in version 3.10.

Macros to extract fields from time objects. The argument must be an instance of `PyDateTime_Time`, including subclasses. The argument must not be `NULL`, and the type is not checked:

int **PyDateTime_TIME_GET_HOUR**(*PyDateTime_Time* *o)

Return the hour, as an int from 0 through 23.

int **PyDateTime_TIME_GET_MINUTE**(*PyDateTime_Time* *o)

Return the minute, as an int from 0 through 59.

int **PyDateTime_TIME_GET_SECOND**(*PyDateTime_Time* *o)

Return the second, as an int from 0 through 59.

int **PyDateTime_TIME_GET_MICROSECOND**(*PyDateTime_Time* *o)

Return the microsecond, as an int from 0 through 999999.

int **PyDateTime_TIME_GET_FOLD**(*PyDateTime_Time* *o)

Return the fold, as an int from 0 through 1.

Added in version 3.6.

PyObject ***PyDateTime_TIME_GET_TZINFO**(*PyDateTime_Time* *o)

Return the tzinfo (which may be `None`).

Added in version 3.10.

Macros to extract fields from time delta objects. The argument must be an instance of `PyDateTime_Delta`, including subclasses. The argument must not be `NULL`, and the type is not checked:

int **PyDateTime_DELTA_GET_DAYS**(*PyDateTime_Delta* *o)

Return the number of days, as an int from -999999999 to 999999999.

Added in version 3.3.

int **PyDateTime_DELTA_GET_SECONDS**(*PyDateTime_Delta* *o)

Return the number of seconds, as an int from 0 through 86399.

Added in version 3.3.

int **PyDateTime_DELTA_GET_MICROSECONDS**(*PyDateTime_Delta* *o)

Return the number of microseconds, as an int from 0 through 999999.

Added in version 3.3.

Macros for the convenience of modules implementing the DB API:

PyObject ***PyDateTime_FromTimestamp**(*PyObject* *args)

Wartość zwracana: nowa referencja. Create and return a new `datetime.datetime` object given an argument tuple suitable for passing to `datetime.datetime.fromtimestamp()`.

PyObject ***PyDate_FromTimestamp**(*PyObject* *args)

Wartość zwracana: nowa referencja. Create and return a new `datetime.date` object given an argument tuple suitable for passing to `datetime.date.fromtimestamp()`.

8.6.14 Objects for Type Hinting

Various built-in types for type hinting are provided. Currently, two types exist – GenericAlias and Union. Only `GenericAlias` is exposed to C.

PyObject ***Py_GenericAlias**(*PyObject* *origin, *PyObject* *args)

Część stabilnego ABI *od wersji 3.9.* Create a GenericAlias object. Equivalent to calling the Python class `types.GenericAlias`. The *origin* and *args* arguments set the `GenericAlias`«s `__origin__` and `__args__` attributes respectively. *origin* should be a `PyTypeObject*`, and *args* can be a `PyTupleObject*` or any `PyObject*`. If *args* passed is not a tuple, a 1-tuple is automatically constructed and `__args__` is set to `(args,)`. Minimal checking is done for the arguments, so the function will succeed even if *origin* is not a type. The `GenericAlias`«s `__parameters__` attribute is constructed lazily from `__args__`. On failure, an exception is raised and `NULL` is returned.

Here's an example of how to make an extension type generic:

```
...
static PyMethodDef my_obj_methods[] = {
    // Other methods.
    ...
    {"__class_getitem__", Py_GenericAlias, METH_O|METH_CLASS, "See PEP 585"}
    ...
}
```

Zobacz także

The data model method `__class_getitem__()`.

Added in version 3.9.

PyTypeObject **`Py_GenericAliasType`**

Część stabilnego ABI *od wersji 3.9.* The C type of the object returned by `Py_GenericAlias()`. Equivalent to `types.GenericAlias` in Python.

Added in version 3.9.

ROZDZIAŁ 9

Initialization, Finalization, and Threads

See *Python Initialization Configuration* for details on how to configure the interpreter prior to initialization.

9.1 Before Python Initialization

In an application embedding Python, the `Py_Initialize()` function must be called before using any other Python/C API functions; with the exception of a few functions and the *global configuration variables*.

The following functions can be safely called before Python is initialized:

- Functions that initialize the interpreter:
  - `Py_Initialize()`
  - `Py_InitializeEx()`
  - `Py_InitializeFromConfig()`
  - `Py_BytesMain()`
  - `Py_Main()`
  - the runtime pre-initialization functions covered in *Python Initialization Configuration*
- Configuration functions:
  - `PyImport_AppendInittab()`
  - `PyImport_ExtendInittab()`
  - `PyInitFrozenExtensions()`
  - `PyMem_SetAllocator()`
  - `PyMem_SetupDebugHooks()`
  - `PyObject_SetArenaAllocator()`
  - `Py_SetProgramName()`
  - `Py_SetPythonHome()`
  - `PySys_ResetWarnOptions()`
  - the configuration functions covered in *Python Initialization Configuration*
- Informative functions:

  - `Py_IsInitialized()`
  - `PyMem_GetAllocator()`
  - `PyObject_GetArenaAllocator()`
  - `Py_GetBuildInfo()`
  - `Py_GetCompiler()`
  - `Py_GetCopyright()`
  - `Py_GetPlatform()`
  - `Py_GetVersion()`
  - `Py_IsInitialized()`
- Utilities:
  - `Py_DecodeLocale()`
  - the status reporting and utility functions covered in *Python Initialization Configuration*
- Memory allocators:
  - `PyMem_RawMalloc()`
  - `PyMem_RawRealloc()`
  - `PyMem_RawCalloc()`
  - `PyMem_RawFree()`
- Synchronization:
  - `PyMutex_Lock()`
  - `PyMutex_Unlock()`

Informacja

Despite their apparent similarity to some of the functions listed above, the following functions **should not be called** before the interpreter has been initialized: `Py_EncodeLocale()`, `Py_GetPath()`, `Py_GetPrefix()`, `Py_GetExecPrefix()`, `Py_GetProgramFullPath()`, `Py_GetPythonHome()`, `Py_GetProgramName()`, `PyEval_InitThreads()`, and `Py_RunMain()`.

9.2 Global configuration variables

Python has variables for the global configuration to control different features and options. By default, these flags are controlled by command line options.

When a flag is set by an option, the value of the flag is the number of times that the option was set. For example, `-b` sets `Py_BytesWarningFlag` to 1 and `-bb` sets `Py_BytesWarningFlag` to 2.

int **`Py_BytesWarningFlag`**

This API is kept for backward compatibility: setting `PyConfig.bytes_warning` should be used instead, see *Python Initialization Configuration*.

Issue a warning when comparing `bytes` or `bytearray` with `str` or `bytes` with `int`. Issue an error if greater or equal to `2`.

Set by the `-b` option.

Deprecated since version 3.12, will be removed in version 3.14.

int **Py_DebugFlag**

This API is kept for backward compatibility: setting `PyConfig.parser_debug` should be used instead, see *Python Initialization Configuration*.

Turn on parser debugging output (for expert only, depending on compilation options).

Set by the `-d` option and the `PYTHONDEBUG` environment variable.

Deprecated since version 3.12, will be removed in version 3.14.

int **Py_DontWriteBytecodeFlag**

This API is kept for backward compatibility: setting `PyConfig.write_bytecode` should be used instead, see *Python Initialization Configuration*.

If set to non-zero, Python won't try to write `.pyc` files on the import of source modules.

Set by the `-B` option and the `PYTHONDONTWRITEBYTECODE` environment variable.

Deprecated since version 3.12, will be removed in version 3.14.

int **Py_FrozenFlag**

This API is kept for backward compatibility: setting `PyConfig.pathconfig_warnings` should be used instead, see *Python Initialization Configuration*.

Suppress error messages when calculating the module search path in `Py_GetPath()`.

Private flag used by `_freeze_module` and `frozenmain` programs.

Deprecated since version 3.12, will be removed in version 3.14.

int **Py_HashRandomizationFlag**

This API is kept for backward compatibility: setting `PyConfig.hash_seed` and `PyConfig.use_hash_seed` should be used instead, see *Python Initialization Configuration*.

Set to `1` if the `PYTHONHASHSEED` environment variable is set to a non-empty string.

If the flag is non-zero, read the `PYTHONHASHSEED` environment variable to initialize the secret hash seed.

Deprecated since version 3.12, will be removed in version 3.14.

int **Py_IgnoreEnvironmentFlag**

This API is kept for backward compatibility: setting `PyConfig.use_environment` should be used instead, see *Python Initialization Configuration*.

Ignore all `PYTHON*` environment variables, e.g. `PYTHONPATH` and `PYTHONHOME`, that might be set.

Set by the `-E` and `-I` options.

Deprecated since version 3.12, will be removed in version 3.14.

int **Py_InspectFlag**

This API is kept for backward compatibility: setting `PyConfig.inspect` should be used instead, see *Python Initialization Configuration*.

When a script is passed as first argument or the `-c` option is used, enter interactive mode after executing the script or the command, even when `sys.stdin` does not appear to be a terminal.

Set by the `-i` option and the `PYTHONINSPECT` environment variable.

Deprecated since version 3.12, will be removed in version 3.14.

int **Py_InteractiveFlag**

This API is kept for backward compatibility: setting `PyConfig.interactive` should be used instead, see *Python Initialization Configuration*.

Set by the `-i` option.

Niezalecane od wersji 3.12.

int **Py_IsolatedFlag**

This API is kept for backward compatibility: setting `PyConfig.isolated` should be used instead, see *Python Initialization Configuration*.

Run Python in isolated mode. In isolated mode `sys.path` contains neither the script's directory nor the user's site-packages directory.

Set by the `-I` option.

Added in version 3.4.

Deprecated since version 3.12, will be removed in version 3.14.

int **Py_LegacyWindowsFSEncodingFlag**

This API is kept for backward compatibility: setting `PyPreConfig.legacy_windows_fs_encoding` should be used instead, see *Python Initialization Configuration*.

If the flag is non-zero, use the `mbcs` encoding with `replace` error handler, instead of the UTF-8 encoding with `surrogatepass` error handler, for the *filesystem encoding and error handler*.

Set to `1` if the `PYTHONLEGACYWINDOWSFSENCODING` environment variable is set to a non-empty string.

See **PEP 529** for more details.

Availability: Windows.

Deprecated since version 3.12, will be removed in version 3.14.

int **Py_LegacyWindowsStdioFlag**

This API is kept for backward compatibility: setting `PyConfig.legacy_windows_stdio` should be used instead, see *Python Initialization Configuration*.

If the flag is non-zero, use `io.FileIO` instead of `io._WindowsConsoleIO` for `sys` standard streams.

Set to `1` if the `PYTHONLEGACYWINDOWSSTDIO` environment variable is set to a non-empty string.

See **PEP 528** for more details.

Availability: Windows.

Deprecated since version 3.12, will be removed in version 3.14.

int **Py_NoSiteFlag**

This API is kept for backward compatibility: setting `PyConfig.site_import` should be used instead, see *Python Initialization Configuration*.

Disable the import of the module `site` and the site-dependent manipulations of `sys.path` that it entails. Also disable these manipulations if `site` is explicitly imported later (call `site.main()` if you want them to be triggered).

Set by the `-S` option.

Deprecated since version 3.12, will be removed in version 3.14.

int **Py_NoUserSiteDirectory**

This API is kept for backward compatibility: setting `PyConfig.user_site_directory` should be used instead, see *Python Initialization Configuration*.

Don't add the `user site-packages directory` to `sys.path`.

Set by the `-s` and `-I` options, and the `PYTHONNOUSERSITE` environment variable.

Deprecated since version 3.12, will be removed in version 3.14.

int **Py_OptimizeFlag**

This API is kept for backward compatibility: setting `PyConfig.optimization_level` should be used instead, see *Python Initialization Configuration*.

Set by the `-O` option and the `PYTHONOPTIMIZE` environment variable.

Deprecated since version 3.12, will be removed in version 3.14.

int **Py_QuietFlag**

This API is kept for backward compatibility: setting `PyConfig.quiet` should be used instead, see *Python Initialization Configuration*.

Don't display the copyright and version messages even in interactive mode.

Set by the `-q` option.

Added in version 3.2.

Deprecated since version 3.12, will be removed in version 3.14.

int **Py_UnbufferedStdioFlag**

This API is kept for backward compatibility: setting `PyConfig.buffered_stdio` should be used instead, see *Python Initialization Configuration*.

Force the stdout and stderr streams to be unbuffered.

Set by the `-u` option and the `PYTHONUNBUFFERED` environment variable.

Deprecated since version 3.12, will be removed in version 3.14.

int **Py_VerboseFlag**

This API is kept for backward compatibility: setting `PyConfig.verbose` should be used instead, see *Python Initialization Configuration*.

Print a message each time a module is initialized, showing the place (filename or built-in module) from which it is loaded. If greater or equal to `2`, print a message for each file that is checked for when searching for a module. Also provides information on module cleanup at exit.

Set by the `-v` option and the `PYTHONVERBOSE` environment variable.

Deprecated since version 3.12, will be removed in version 3.14.

9.3 Initializing and finalizing the interpreter

void **Py_Initialize**()

Część stabilnego ABI. Initialize the Python interpreter. In an application embedding Python, this should be called before using any other Python/C API functions; see *Before Python Initialization* for the few exceptions.

This initializes the table of loaded modules (`sys.modules`), and creates the fundamental modules `builtins`, `__main__` and `sys`. It also initializes the module search path (`sys.path`). It does not set `sys.argv`; use the *Python Initialization Configuration* API for that. This is a no-op when called for a second time (without calling `Py_FinalizeEx()` first). There is no return value; it is a fatal error if the initialization fails.

Use `Py_InitializeFromConfig()` to customize the *Python Initialization Configuration*.

> **Informacja**
>
> On Windows, changes the console mode from `O_TEXT` to `O_BINARY`, which will also affect non-Python uses of the console using the C Runtime.

void **Py_InitializeEx**(int initsigs)

Część stabilnego ABI. This function works like `Py_Initialize()` if *initsigs* is `1`. If *initsigs* is `0`, it skips initialization registration of signal handlers, which may be useful when CPython is embedded as part of a larger application.

Use `Py_InitializeFromConfig()` to customize the *Python Initialization Configuration*.

PyStatus **Py_InitializeFromConfig**(const *PyConfig* *config)

Initialize Python from *config* configuration, as described in *Initialization with PyConfig*.

See the *Python Initialization Configuration* section for details on pre-initializing the interpreter, populating the runtime configuration structure, and querying the returned status structure.

int **Py_IsInitialized**()

Część stabilnego ABI. Return true (nonzero) when the Python interpreter has been initialized, false (zero) if not. After `Py_FinalizeEx()` is called, this returns false until `Py_Initialize()` is called again.

int **Py_IsFinalizing**()

Część stabilnego ABI *od wersji 3.13.* Return true (non-zero) if the main Python interpreter is *shutting down*. Return false (zero) otherwise.

Added in version 3.13.

int **Py_FinalizeEx**()

Część stabilnego ABI *od wersji 3.6.* Undo all initializations made by `Py_Initialize()` and subsequent use of Python/C API functions, and destroy all sub-interpreters (see `Py_NewInterpreter()` below) that were created and not yet destroyed since the last call to `Py_Initialize()`. Ideally, this frees all memory allocated by the Python interpreter. This is a no-op when called for a second time (without calling `Py_Initialize()` again first).

Since this is the reverse of `Py_Initialize()`, it should be called in the same thread with the same interpreter active. That means the main thread and the main interpreter. This should never be called while `Py_RunMain()` is running.

Normally the return value is `0`. If there were errors during finalization (flushing buffered data), `-1` is returned.

This function is provided for a number of reasons. An embedding application might want to restart Python without having to restart the application itself. An application that has loaded the Python interpreter from a dynamically loadable library (or DLL) might want to free all memory allocated by Python before unloading the DLL. During a hunt for memory leaks in an application a developer might want to free all memory allocated by Python before exiting from the application.

Bugs and caveats: The destruction of modules and objects in modules is done in random order; this may cause destructors (`__del__()` methods) to fail when they depend on other objects (even functions) or modules. Dynamically loaded extension modules loaded by Python are not unloaded. Small amounts of memory allocated by the Python interpreter may not be freed (if you find a leak, please report it). Memory tied up in circular references between objects is not freed. Some memory allocated by extension modules may not be freed. Some extensions may not work properly if their initialization routine is called more than once; this can happen if an application calls `Py_Initialize()` and `Py_FinalizeEx()` more than once.

Raises an auditing event `cpython._PySys_ClearAuditHooks` with no arguments.

Added in version 3.6.

void **Py_Finalize**()

Część stabilnego ABI. This is a backwards-compatible version of `Py_FinalizeEx()` that disregards the return value.

int **Py_BytesMain**(int argc, char **argv)

Część stabilnego ABI *od wersji 3.8.* Similar to `Py_Main()` but *argv* is an array of bytes strings, allowing the calling application to delegate the text decoding step to the CPython runtime.

Added in version 3.8.

int **Py_Main**(int argc, wchar_t **argv)

Część stabilnego ABI. The main program for the standard interpreter, encapsulating a full initialization/finalization cycle, as well as additional behaviour to implement reading configurations settings from the environment and command line, and then executing `__main__` in accordance with using-on-cmdline.

This is made available for programs which wish to support the full CPython command line interface, rather than just embedding a Python runtime in a larger application.

The *argc* and *argv* parameters are similar to those which are passed to a C program's `main()` function, except that the *argv* entries are first converted to `wchar_t` using `Py_DecodeLocale()`. It is also important to note that the argument list entries may be modified to point to strings other than those passed in (however, the contents of the strings pointed to by the argument list are not modified).

The return value will be `0` if the interpreter exits normally (i.e., without an exception), `1` if the interpreter exits due to an exception, or `2` if the argument list does not represent a valid Python command line.

Note that if an otherwise unhandled `SystemExit` is raised, this function will not return `1`, but exit the process, as long as `Py_InspectFlag` is not set. If `Py_InspectFlag` is set, execution will drop into the interactive Python prompt, at which point a second otherwise unhandled `SystemExit` will still exit the process, while any other means of exiting will set the return value as described above.

In terms of the CPython runtime configuration APIs documented in the *runtime configuration* section (and without accounting for error handling), `Py_Main` is approximately equivalent to:

```
PyConfig config;
PyConfig_InitPythonConfig(&config);
PyConfig_SetArgv(&config, argc, argv);
Py_InitializeFromConfig(&config);
PyConfig_Clear(&config);

Py_RunMain();
```

In normal usage, an embedding application will call this function *instead* of calling `Py_Initialize()`, `Py_InitializeEx()` or `Py_InitializeFromConfig()` directly, and all settings will be applied as described elsewhere in this documentation. If this function is instead called *after* a preceding runtime initialization API call, then exactly which environmental and command line configuration settings will be updated is version dependent (as it depends on which settings correctly support being modified after they have already been set once when the runtime was first initialized).

int **Py_RunMain**(void)

Executes the main module in a fully configured CPython runtime.

Executes the command (`PyConfig.run_command`), the script (`PyConfig.run_filename`) or the module (`PyConfig.run_module`) specified on the command line or in the configuration. If none of these values are set, runs the interactive Python prompt (REPL) using the `__main__` module's global namespace.

If `PyConfig.inspect` is not set (the default), the return value will be `0` if the interpreter exits normally (that is, without raising an exception), or `1` if the interpreter exits due to an exception. If an otherwise unhandled `SystemExit` is raised, the function will immediately exit the process instead of returning `1`.

If `PyConfig.inspect` is set (such as when the `-i` option is used), rather than returning when the interpreter exits, execution will instead resume in an interactive Python prompt (REPL) using the `__main__` module's global namespace. If the interpreter exited with an exception, it is immediately raised in the REPL session. The function return value is then determined by the way the *REPL session* terminates: returning `0` if the session terminates without raising an unhandled exception, exiting immediately for an unhandled `SystemExit`, and returning `1` for any other unhandled exception.

This function always finalizes the Python interpreter regardless of whether it returns a value or immediately exits the process due to an unhandled `SystemExit` exception.

See *Python Configuration* for an example of a customized Python that always runs in isolated mode using `Py_RunMain()`.

9.4 Process-wide parameters

void **Py_SetProgramName**(const wchar_t *name)

Część stabilnego ABI. This API is kept for backward compatibility: setting `PyConfig.program_name` should be used instead, see *Python Initialization Configuration*.

This function should be called before `Py_Initialize()` is called for the first time, if it is called at all. It tells the interpreter the value of the `argv[0]` argument to the `main()` function of the program (converted to wide characters). This is used by `Py_GetPath()` and some other functions below to find the Python run-time libraries relative to the interpreter executable. The default value is `'python'`. The argument should point to a zero-terminated wide character string in static storage whose contents will not change for the duration of the program's execution. No code in the Python interpreter will change the contents of this storage.

Use `Py_DecodeLocale()` to decode a bytes string to get a `wchar_*` string.

Niezalecane od wersji 3.11.

wchar_t ***Py_GetProgramName**()

Część stabilnego ABI. Return the program name set with `PyConfig.program_name`, or the default. The returned string points into static storage; the caller should not modify its value.

This function should not be called before `Py_Initialize()`, otherwise it returns `NULL`.

Zmienione w wersji 3.10: It now returns `NULL` if called before `Py_Initialize()`.

Deprecated since version 3.13, will be removed in version 3.15: Get `sys.executable` instead.

wchar_t ***Py_GetPrefix**()

Część stabilnego ABI. Return the *prefix* for installed platform-independent files. This is derived through a number of complicated rules from the program name set with `PyConfig.program_name` and some environment variables; for example, if the program name is `'/usr/local/bin/python'`, the prefix is `'/usr/local'`. The returned string points into static storage; the caller should not modify its value. This corresponds to the **prefix** variable in the top-level `Makefile` and the `--prefix` argument to the **configure** script at build time. The value is available to Python code as `sys.prefix`. It is only useful on Unix. See also the next function.

This function should not be called before `Py_Initialize()`, otherwise it returns `NULL`.

Zmienione w wersji 3.10: It now returns `NULL` if called before `Py_Initialize()`.

Deprecated since version 3.13, will be removed in version 3.15: Get `sys.prefix` instead.

wchar_t ***Py_GetExecPrefix**()

Część stabilnego ABI. Return the *exec-prefix* for installed platform-*dependent* files. This is derived through a number of complicated rules from the program name set with `PyConfig.program_name` and some environment variables; for example, if the program name is `'/usr/local/bin/python'`, the exec-prefix is `'/usr/local'`. The returned string points into static storage; the caller should not modify its value. This corresponds to the **exec_prefix** variable in the top-level `Makefile` and the `--exec-prefix` argument to the **configure** script at build time. The value is available to Python code as `sys.exec_prefix`. It is only useful on Unix.

Background: The exec-prefix differs from the prefix when platform dependent files (such as executables and shared libraries) are installed in a different directory tree. In a typical installation, platform dependent files may be installed in the `/usr/local/plat` subtree while platform independent may be installed in `/usr/local`.

Generally speaking, a platform is a combination of hardware and software families, e.g. Sparc machines running the Solaris 2.x operating system are considered the same platform, but Intel machines running Solaris 2.x are another platform, and Intel machines running Linux are yet another platform. Different major revisions of the same operating system generally also form different platforms. Non-Unix operating systems are a different story; the installation strategies on those systems are so different that the prefix and exec-prefix are meaningless, and set to the empty string. Note that compiled Python bytecode files are platform independent (but not independent from the Python version by which they were compiled!).

System administrators will know how to configure the **mount** or **automount** programs to share `/usr/local` between platforms while having `/usr/local/plat` be a different filesystem for each platform.

This function should not be called before `Py_Initialize()`, otherwise it returns `NULL`.

Zmienione w wersji 3.10: It now returns `NULL` if called before `Py_Initialize()`.

Deprecated since version 3.13, will be removed in version 3.15: Get `sys.exec_prefix` instead.

wchar_t ***Py_GetProgramFullPath**()

Część stabilnego ABI. Return the full program name of the Python executable; this is computed as a side-effect of deriving the default module search path from the program name (set by `PyConfig.program_name`). The returned string points into static storage; the caller should not modify its value. The value is available to Python code as `sys.executable`.

This function should not be called before `Py_Initialize()`, otherwise it returns `NULL`.

Zmienione w wersji 3.10: It now returns `NULL` if called before `Py_Initialize()`.

Deprecated since version 3.13, will be removed in version 3.15: Get `sys.executable` instead.

wchar_t ***Py_GetPath**()

Część stabilnego ABI. Return the default module search path; this is computed from the program name (set by `PyConfig.program_name`) and some environment variables. The returned string consists of a series of directory names separated by a platform dependent delimiter character. The delimiter character is `':'` on Unix and macOS, `';'` on Windows. The returned string points into static storage; the caller should not modify its value. The list `sys.path` is initialized with this value on interpreter startup; it can be (and usually is) modified later to change the search path for loading modules.

This function should not be called before `Py_Initialize()`, otherwise it returns `NULL`.

Zmienione w wersji 3.10: It now returns `NULL` if called before `Py_Initialize()`.

Deprecated since version 3.13, will be removed in version 3.15: Get `sys.path` instead.

const char ***Py_GetVersion**()

Część stabilnego ABI. Return the version of this Python interpreter. This is a string that looks something like

```
"3.0a5+ (py3k:63103M, May 12 2008, 00:53:55) \n[GCC 4.2.3]"
```

The first word (up to the first space character) is the current Python version; the first characters are the major and minor version separated by a period. The returned string points into static storage; the caller should not modify its value. The value is available to Python code as `sys.version`.

See also the `Py_Version` constant.

const char ***Py_GetPlatform**()

Część stabilnego ABI. Return the platform identifier for the current platform. On Unix, this is formed from the „official" name of the operating system, converted to lower case, followed by the major revision number; e.g., for Solaris 2.x, which is also known as SunOS 5.x, the value is `'sunos5'`. On macOS, it is `'darwin'`. On Windows, it is `'win'`. The returned string points into static storage; the caller should not modify its value. The value is available to Python code as `sys.platform`.

const char ***Py_GetCopyright**()

Część stabilnego ABI. Return the official copyright string for the current Python version, for example

```
'Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam'
```

The returned string points into static storage; the caller should not modify its value. The value is available to Python code as `sys.copyright`.

const char ***Py_GetCompiler**()

Część stabilnego ABI. Return an indication of the compiler used to build the current Python version, in square brackets, for example:

```
"[GCC 2.7.2.2]"
```

The returned string points into static storage; the caller should not modify its value. The value is available to Python code as part of the variable `sys.version`.

const char ***Py_GetBuildInfo**()

Część stabilnego ABI. Return information about the sequence number and build date and time of the current Python interpreter instance, for example

```
"#67, Aug  1 1997, 22:34:28"
```

The returned string points into static storage; the caller should not modify its value. The value is available to Python code as part of the variable `sys.version`.

void **PySys_SetArgvEx**(int argc, wchar_t **argv, int updatepath)

Część stabilnego ABI. This API is kept for backward compatibility: setting `PyConfig.argv`, `PyConfig.parse_argv` and `PyConfig.safe_path` should be used instead, see *Python Initialization Configuration*.

Set `sys.argv` based on *argc* and *argv*. These parameters are similar to those passed to the program's `main()` function with the difference that the first entry should refer to the script file to be executed rather than the executable hosting the Python interpreter. If there isn't a script that will be run, the first entry in *argv* can be an empty string. If this function fails to initialize `sys.argv`, a fatal condition is signalled using `Py_FatalError()`.

If *updatepath* is zero, this is all the function does. If *updatepath* is non-zero, the function also modifies `sys.path` according to the following algorithm:

- If the name of an existing script is passed in `argv[0]`, the absolute path of the directory where the script is located is prepended to `sys.path`.
- Otherwise (that is, if *argc* is `0` or `argv[0]` doesn't point to an existing file name), an empty string is prepended to `sys.path`, which is the same as prepending the current working directory (`".`").

Use `Py_DecodeLocale()` to decode a bytes string to get a `wchar_*` string.

See also `PyConfig.orig_argv` and `PyConfig.argv` members of the *Python Initialization Configuration*.

Informacja

It is recommended that applications embedding the Python interpreter for purposes other than executing a single script pass `0` as *updatepath*, and update `sys.path` themselves if desired. See **CVE 2008-5983**.

On versions before 3.1.3, you can achieve the same effect by manually popping the first `sys.path` element after having called `PySys_SetArgv()`, for example using:

```
PyRun_SimpleString("import sys; sys.path.pop(0)\n");
```

Added in version 3.1.3.

Niezalecane od wersji 3.11.

void **PySys_SetArgv**(int argc, wchar_t **argv)

Część stabilnego ABI. This API is kept for backward compatibility: setting `PyConfig.argv` and `PyConfig.parse_argv` should be used instead, see *Python Initialization Configuration*.

This function works like `PySys_SetArgvEx()` with *updatepath* set to `1` unless the **python** interpreter was started with the `-I`.

Use `Py_DecodeLocale()` to decode a bytes string to get a `wchar_*` string.

See also `PyConfig.orig_argv` and `PyConfig.argv` members of the *Python Initialization Configuration*.

Zmienione w wersji 3.4: The *updatepath* value depends on `-I`.

Niezalecane od wersji 3.11.

void **Py_SetPythonHome**(const wchar_t *home)

Część stabilnego ABI. This API is kept for backward compatibility: setting `PyConfig.home` should be used instead, see *Python Initialization Configuration*.

Set the default „home" directory, that is, the location of the standard Python libraries. See `PYTHONHOME` for the meaning of the argument string.

The argument should point to a zero-terminated character string in static storage whose contents will not change for the duration of the program's execution. No code in the Python interpreter will change the contents of this storage.

Use `Py_DecodeLocale()` to decode a bytes string to get a `wchar_*` string.

Niezalecane od wersji 3.11.

wchar_t ***Py_GetPythonHome**()

Część stabilnego ABI. Return the default „home", that is, the value set by `PyConfig.home`, or the value of the `PYTHONHOME` environment variable if it is set.

This function should not be called before `Py_Initialize()`, otherwise it returns `NULL`.

Zmienione w wersji 3.10: It now returns `NULL` if called before `Py_Initialize()`.

Deprecated since version 3.13, will be removed in version 3.15: Get `PyConfig.home` or `PYTHONHOME` environment variable instead.

9.5 Thread State and the Global Interpreter Lock

The Python interpreter is not fully thread-safe. In order to support multi-threaded Python programs, there's a global lock, called the *global interpreter lock* or *GIL*, that must be held by the current thread before it can safely access Python objects. Without the lock, even the simplest operations could cause problems in a multi-threaded program: for example, when two threads simultaneously increment the reference count of the same object, the reference count could end up being incremented only once instead of twice.

Therefore, the rule exists that only the thread that has acquired the *GIL* may operate on Python objects or call Python/C API functions. In order to emulate concurrency of execution, the interpreter regularly tries to switch threads (see `sys.setswitchinterval()`). The lock is also released around potentially blocking I/O operations like reading or writing a file, so that other Python threads can run in the meantime.

The Python interpreter keeps some thread-specific bookkeeping information inside a data structure called `PyThreadState`. There's also one global variable pointing to the current `PyThreadState`: it can be retrieved using `PyThreadState_Get()`.

9.5.1 Releasing the GIL from extension code

Most extension code manipulating the *GIL* has the following simple structure:

```
Save the thread state in a local variable.
Release the global interpreter lock.
... Do some blocking I/O operation ...
Reacquire the global interpreter lock.
Restore the thread state from the local variable.
```

This is so common that a pair of macros exists to simplify it:

```
Py_BEGIN_ALLOW_THREADS
... Do some blocking I/O operation ...
Py_END_ALLOW_THREADS
```

The `Py_BEGIN_ALLOW_THREADS` macro opens a new block and declares a hidden local variable; the `Py_END_ALLOW_THREADS` macro closes the block.

The block above expands to the following code:

```
PyThreadState *_save;

_save = PyEval_SaveThread();
... Do some blocking I/O operation ...
PyEval_RestoreThread(_save);
```

Here is how these functions work: the global interpreter lock is used to protect the pointer to the current thread state. When releasing the lock and saving the thread state, the current thread state pointer must be retrieved before the lock is released (since another thread could immediately acquire the lock and store its own thread state in the global variable). Conversely, when acquiring the lock and restoring the thread state, the lock must be acquired before storing the thread state pointer.

> **Informacja**
>
> Calling system I/O functions is the most common use case for releasing the GIL, but it can also be useful before calling long-running computations which don't need access to Python objects, such as compression or cryptographic functions operating over memory buffers. For example, the standard `zlib` and `hashlib` modules release the GIL when compressing or hashing data.

9.5.2 Non-Python created threads

When threads are created using the dedicated Python APIs (such as the `threading` module), a thread state is automatically associated to them and the code showed above is therefore correct. However, when threads are created from C (for example by a third-party library with its own thread management), they don't hold the GIL, nor is there a thread state structure for them.

If you need to call Python code from these threads (often this will be part of a callback API provided by the aforementioned third-party library), you must first register these threads with the interpreter by creating a thread state data structure, then acquiring the GIL, and finally storing their thread state pointer, before you can start using the Python/C API. When you are done, you should reset the thread state pointer, release the GIL, and finally free the thread state data structure.

The *PyGILState_Ensure()* and *PyGILState_Release()* functions do all of the above automatically. The typical idiom for calling into Python from a C thread is:

```
PyGILState_STATE gstate;
gstate = PyGILState_Ensure();

/* Perform Python actions here. */
result = CallSomeFunction();
/* evaluate result or handle exception */

/* Release the thread. No Python API allowed beyond this point. */
PyGILState_Release(gstate);
```

Note that the `PyGILState_*` functions assume there is only one global interpreter (created automatically by *Py_Initialize()*). Python supports the creation of additional interpreters (using *Py_NewInterpreter()*), but mixing multiple interpreters and the `PyGILState_*` API is unsupported.

9.5.3 Cautions about fork()

Another important thing to note about threads is their behaviour in the face of the C `fork()` call. On most systems with `fork()`, after a process forks only the thread that issued the fork will exist. This has a concrete impact both on how locks must be handled and on all stored state in CPython's runtime.

The fact that only the „current" thread remains means any locks held by other threads will never be released. Python solves this for `os.fork()` by acquiring the locks it uses internally before the fork, and releasing them afterwards. In addition, it resets any lock-objects in the child. When extending or embedding Python, there is no way to inform Python of additional (non-Python) locks that need to be acquired before or reset after a fork. OS facilities such as `pthread_atfork()` would need to be used to accomplish the same thing. Additionally, when extending or embedding Python, calling `fork()` directly rather than through `os.fork()` (and returning to or calling into Python) may result in a deadlock by one of Python's internal locks being held by a thread that is defunct after the fork. *PyOS_AfterFork_Child()* tries to reset the necessary locks, but is not always able to.

The fact that all other threads go away also means that CPython's runtime state there must be cleaned up properly,

which `os.fork()` does. This means finalizing all other `PyThreadState` objects belonging to the current interpreter and all other `PyInterpreterState` objects. Due to this and the special nature of the *„main" interpreter*, `fork()` should only be called in that interpreter's „main" thread, where the CPython global runtime was originally initialized. The only exception is if `exec()` will be called immediately after.

9.5.4 High-level API

These are the most commonly used types and functions when writing C extension code, or when embedding the Python interpreter:

type **PyInterpreterState**

Część ograniczonego API *(jako nieprzezroczysta struktura).* This data structure represents the state shared by a number of cooperating threads. Threads belonging to the same interpreter share their module administration and a few other internal items. There are no public members in this structure.

Threads belonging to different interpreters initially share nothing, except process state like available memory, open file descriptors and such. The global interpreter lock is also shared by all threads, regardless of to which interpreter they belong.

type **PyThreadState**

Część ograniczonego API *(jako nieprzezroczysta struktura).* This data structure represents the state of a single thread. The only public data member is:

PyInterpreterState ***interp**

This thread's interpreter state.

void **PyEval_InitThreads**()

Część stabilnego ABI. Deprecated function which does nothing.

In Python 3.6 and older, this function created the GIL if it didn't exist.

Zmienione w wersji 3.9: The function now does nothing.

Zmienione w wersji 3.7: This function is now called by `Py_Initialize()`, so you don't have to call it yourself anymore.

Zmienione w wersji 3.2: This function cannot be called before `Py_Initialize()` anymore.

Niezalecane od wersji 3.9.

PyThreadState ***PyEval_SaveThread**()

Część stabilnego ABI. Release the global interpreter lock (if it has been created) and reset the thread state to `NULL`, returning the previous thread state (which is not `NULL`). If the lock has been created, the current thread must have acquired it.

void **PyEval_RestoreThread**(*PyThreadState* *tstate)

Część stabilnego ABI. Acquire the global interpreter lock (if it has been created) and set the thread state to *tstate*, which must not be `NULL`. If the lock has been created, the current thread must not have acquired it, otherwise deadlock ensues.

> **Informacja**
>
> Calling this function from a thread when the runtime is finalizing will terminate the thread, even if the thread was not created by Python. You can use `Py_IsFinalizing()` or `sys.is_finalizing()` to check if the interpreter is in process of being finalized before calling this function to avoid unwanted termination.

PyThreadState ***PyThreadState_Get**()

Część stabilnego ABI. Return the current thread state. The global interpreter lock must be held. When the current thread state is `NULL`, this issues a fatal error (so that the caller needn't check for `NULL`).

See also `PyThreadState_GetUnchecked()`.

PyThreadState ***PyThreadState_GetUnchecked**()

Similar to `PyThreadState_Get()`, but don't kill the process with a fatal error if it is NULL. The caller is responsible to check if the result is NULL.

Added in version 3.13: In Python 3.5 to 3.12, the function was private and known as `_PyThreadState_UncheckedGet()`.

PyThreadState ***PyThreadState_Swap**(*PyThreadState* *tstate)

Część stabilnego ABI. Swap the current thread state with the thread state given by the argument *tstate*, which may be `NULL`. The global interpreter lock must be held and is not released.

The following functions use thread-local storage, and are not compatible with sub-interpreters:

PyGILState_STATE **PyGILState_Ensure**()

Część stabilnego ABI. Ensure that the current thread is ready to call the Python C API regardless of the current state of Python, or of the global interpreter lock. This may be called as many times as desired by a thread as long as each call is matched with a call to `PyGILState_Release()`. In general, other thread--related APIs may be used between `PyGILState_Ensure()` and `PyGILState_Release()` calls as long as the thread state is restored to its previous state before the Release(). For example, normal usage of the `Py_BEGIN_ALLOW_THREADS` and `Py_END_ALLOW_THREADS` macros is acceptable.

The return value is an opaque „handle" to the thread state when `PyGILState_Ensure()` was called, and must be passed to `PyGILState_Release()` to ensure Python is left in the same state. Even though recursive calls are allowed, these handles *cannot* be shared - each unique call to `PyGILState_Ensure()` must save the handle for its call to `PyGILState_Release()`.

When the function returns, the current thread will hold the GIL and be able to call arbitrary Python code. Failure is a fatal error.

> **Informacja**
>
> Calling this function from a thread when the runtime is finalizing will terminate the thread, even if the thread was not created by Python. You can use `Py_IsFinalizing()` or `sys.is_finalizing()` to check if the interpreter is in process of being finalized before calling this function to avoid unwanted termination.

void **PyGILState_Release**(PyGILState_STATE)

Część stabilnego ABI. Release any resources previously acquired. After this call, Python's state will be the same as it was prior to the corresponding `PyGILState_Ensure()` call (but generally this state will be unknown to the caller, hence the use of the GILState API).

Every call to `PyGILState_Ensure()` must be matched by a call to `PyGILState_Release()` on the same thread.

PyThreadState ***PyGILState_GetThisThreadState**()

Część stabilnego ABI. Get the current thread state for this thread. May return `NULL` if no GILState API has been used on the current thread. Note that the main thread always has such a thread-state, even if no auto--thread-state call has been made on the main thread. This is mainly a helper/diagnostic function.

int **PyGILState_Check**()

Return `1` if the current thread is holding the GIL and `0` otherwise. This function can be called from any thread at any time. Only if it has had its Python thread state initialized and currently is holding the GIL will it return `1`. This is mainly a helper/diagnostic function. It can be useful for example in callback contexts or memory allocation functions when knowing that the GIL is locked can allow the caller to perform sensitive actions or otherwise behave differently.

Added in version 3.4.

The following macros are normally used without a trailing semicolon; look for example usage in the Python source distribution.

Py_BEGIN_ALLOW_THREADS

Część stabilnego ABI. This macro expands to `{ PyThreadState *_save; _save = PyEval_SaveThread();`. Note that it contains an opening brace; it must be matched with a following `Py_END_ALLOW_THREADS` macro. See above for further discussion of this macro.

Py_END_ALLOW_THREADS

Część stabilnego ABI. This macro expands to `PyEval_RestoreThread(_save); }`. Note that it contains a closing brace; it must be matched with an earlier `Py_BEGIN_ALLOW_THREADS` macro. See above for further discussion of this macro.

Py_BLOCK_THREADS

Część stabilnego ABI. This macro expands to `PyEval_RestoreThread(_save);`: it is equivalent to `Py_END_ALLOW_THREADS` without the closing brace.

Py_UNBLOCK_THREADS

Część stabilnego ABI. This macro expands to `_save = PyEval_SaveThread();`: it is equivalent to `Py_BEGIN_ALLOW_THREADS` without the opening brace and variable declaration.

9.5.5 Low-level API

All of the following functions must be called after `Py_Initialize()`.

Zmienione w wersji 3.7: `Py_Initialize()` now initializes the *GIL*.

PyInterpreterState ***PyInterpreterState_New**()

Część stabilnego ABI. Create a new interpreter state object. The global interpreter lock need not be held, but may be held if it is necessary to serialize calls to this function.

Raises an auditing event `cpython.PyInterpreterState_New` with no arguments.

void **PyInterpreterState_Clear**(*PyInterpreterState* *interp)

Część stabilnego ABI. Reset all information in an interpreter state object. The global interpreter lock must be held.

Raises an auditing event `cpython.PyInterpreterState_Clear` with no arguments.

void **PyInterpreterState_Delete**(*PyInterpreterState* *interp)

Część stabilnego ABI. Destroy an interpreter state object. The global interpreter lock need not be held. The interpreter state must have been reset with a previous call to `PyInterpreterState_Clear()`.

PyThreadState ***PyThreadState_New**(*PyInterpreterState* *interp)

Część stabilnego ABI. Create a new thread state object belonging to the given interpreter object. The global interpreter lock need not be held, but may be held if it is necessary to serialize calls to this function.

void **PyThreadState_Clear**(*PyThreadState* *tstate)

Część stabilnego ABI. Reset all information in a thread state object. The global interpreter lock must be held.

Zmienione w wersji 3.9: This function now calls the `PyThreadState.on_delete` callback. Previously, that happened in `PyThreadState_Delete()`.

void **PyThreadState_Delete**(*PyThreadState* *tstate)

Część stabilnego ABI. Destroy a thread state object. The global interpreter lock need not be held. The thread state must have been reset with a previous call to `PyThreadState_Clear()`.

void **PyThreadState_DeleteCurrent**(void)

Destroy the current thread state and release the global interpreter lock. Like `PyThreadState_Delete()`, the global interpreter lock must be held. The thread state must have been reset with a previous call to `PyThreadState_Clear()`.

PyFrameObject ***PyThreadState_GetFrame**(*PyThreadState* *tstate)

Część stabilnego ABI *od wersji 3.10*. Get the current frame of the Python thread state *tstate*.

Return a *strong reference*. Return `NULL` if no frame is currently executing.

See also `PyEval_GetFrame()`.

tstate must not be `NULL`.

Added in version 3.9.

uint64_t **`PyThreadState_GetID`**(*PyThreadState* *tstate)

Część stabilnego ABI *od wersji 3.10.* Get the unique thread state identifier of the Python thread state *tstate*.

tstate must not be `NULL`.

Added in version 3.9.

PyInterpreterState ***`PyThreadState_GetInterpreter`**(*PyThreadState* *tstate)

Część stabilnego ABI *od wersji 3.10.* Get the interpreter of the Python thread state *tstate*.

tstate must not be `NULL`.

Added in version 3.9.

void **`PyThreadState_EnterTracing`**(*PyThreadState* *tstate)

Suspend tracing and profiling in the Python thread state *tstate*.

Resume them using the `PyThreadState_LeaveTracing()` function.

Added in version 3.11.

void **`PyThreadState_LeaveTracing`**(*PyThreadState* *tstate)

Resume tracing and profiling in the Python thread state *tstate* suspended by the `PyThreadState_EnterTracing()` function.

See also `PyEval_SetTrace()` and `PyEval_SetProfile()` functions.

Added in version 3.11.

PyInterpreterState ***`PyInterpreterState_Get`**(void)

Część stabilnego ABI *od wersji 3.9.* Get the current interpreter.

Issue a fatal error if there no current Python thread state or no current interpreter. It cannot return NULL.

The caller must hold the GIL.

Added in version 3.9.

int64_t **`PyInterpreterState_GetID`**(*PyInterpreterState* *interp)

Część stabilnego ABI *od wersji 3.7.* Return the interpreter's unique ID. If there was any error in doing so then `-1` is returned and an error is set.

The caller must hold the GIL.

Added in version 3.7.

PyObject ***`PyInterpreterState_GetDict`**(*PyInterpreterState* *interp)

Część stabilnego ABI *od wersji 3.8.* Return a dictionary in which interpreter-specific data may be stored. If this function returns `NULL` then no exception has been raised and the caller should assume no interpreter-specific dict is available.

This is not a replacement for `PyModule_GetState()`, which extensions should use to store interpreter--specific state information.

Added in version 3.8.

typedef *PyObject* *(***`_PyFrameEvalFunction`**)(*PyThreadState* *tstate, *_PyInterpreterFrame* *frame, int throwflag)

Type of a frame evaluation function.

The *throwflag* parameter is used by the `throw()` method of generators: if non-zero, handle the current exception.

Zmienione w wersji 3.9: The function now takes a *tstate* parameter.

Zmienione w wersji 3.11: The *frame* parameter changed from `PyFrameObject*` to `_PyInterpreterFrame*`.

_PyFrameEvalFunction **_PyInterpreterState_GetEvalFrameFunc**(*PyInterpreterState* *interp)

Get the frame evaluation function.

See the **PEP 523** „Adding a frame evaluation API to CPython".

Added in version 3.9.

void **_PyInterpreterState_SetEvalFrameFunc**(*PyInterpreterState* *interp, *_PyFrameEvalFunction* eval_frame)

Set the frame evaluation function.

See the **PEP 523** „Adding a frame evaluation API to CPython".

Added in version 3.9.

PyObject ***PyThreadState_GetDict**()

Wartość zwracana: pożyczona referencja. Część stabilnego ABI. Return a dictionary in which extensions can store thread-specific state information. Each extension should use a unique key to use to store state in the dictionary. It is okay to call this function when no current thread state is available. If this function returns `NULL`, no exception has been raised and the caller should assume no current thread state is available.

int **PyThreadState_SetAsyncExc**(unsigned long id, *PyObject* *exc)

Część stabilnego ABI. Asynchronously raise an exception in a thread. The *id* argument is the thread id of the target thread; *exc* is the exception object to be raised. This function does not steal any references to *exc*. To prevent naive misuse, you must write your own C extension to call this. Must be called with the GIL held. Returns the number of thread states modified; this is normally one, but will be zero if the thread id isn't found. If *exc* is `NULL`, the pending exception (if any) for the thread is cleared. This raises no exceptions.

Zmienione w wersji 3.7: The type of the *id* parameter changed from `long` to `unsigned long`.

void **PyEval_AcquireThread**(*PyThreadState* *tstate)

Część stabilnego ABI. Acquire the global interpreter lock and set the current thread state to *tstate*, which must not be `NULL`. The lock must have been created earlier. If this thread already has the lock, deadlock ensues.

> **Informacja**
>
> Calling this function from a thread when the runtime is finalizing will terminate the thread, even if the thread was not created by Python. You can use `Py_IsFinalizing()` or `sys.is_finalizing()` to check if the interpreter is in process of being finalized before calling this function to avoid unwanted termination.

Zmienione w wersji 3.8: Updated to be consistent with `PyEval_RestoreThread()`, `Py_END_ALLOW_THREADS()`, and `PyGILState_Ensure()`, and terminate the current thread if called while the interpreter is finalizing.

`PyEval_RestoreThread()` is a higher-level function which is always available (even when threads have not been initialized).

void **PyEval_ReleaseThread**(*PyThreadState* *tstate)

Część stabilnego ABI. Reset the current thread state to `NULL` and release the global interpreter lock. The lock must have been created earlier and must be held by the current thread. The *tstate* argument, which must not be `NULL`, is only used to check that it represents the current thread state — if it isn't, a fatal error is reported.

`PyEval_SaveThread()` is a higher-level function which is always available (even when threads have not been initialized).

9.6 Sub-interpreter support

While in most uses, you will only embed a single Python interpreter, there are cases where you need to create several independent interpreters in the same process and perhaps even in the same thread. Sub-interpreters allow you to do that.

The „main" interpreter is the first one created when the runtime initializes. It is usually the only Python interpreter in a process. Unlike sub-interpreters, the main interpreter has unique process-global responsibilities like signal handling. It is also responsible for execution during runtime initialization and is usually the active interpreter during runtime finalization. The `PyInterpreterState_Main()` function returns a pointer to its state.

You can switch between sub-interpreters using the `PyThreadState_Swap()` function. You can create and destroy them using the following functions:

type **PyInterpreterConfig**

Structure containing most parameters to configure a sub-interpreter. Its values are used only in `Py_NewInterpreterFromConfig()` and never modified by the runtime.

Added in version 3.12.

Structure fields:

int **use_main_obmalloc**

If this is `0` then the sub-interpreter will use its own „object" allocator state. Otherwise it will use (share) the main interpreter's.

If this is `0` then `check_multi_interp_extensions` must be `1` (non-zero). If this is `1` then `gil` must not be `PyInterpreterConfig_OWN_GIL`.

int **allow_fork**

If this is `0` then the runtime will not support forking the process in any thread where the sub-interpreter is currently active. Otherwise fork is unrestricted.

Note that the `subprocess` module still works when fork is disallowed.

int **allow_exec**

If this is `0` then the runtime will not support replacing the current process via exec (e.g. `os.execv()`) in any thread where the sub-interpreter is currently active. Otherwise exec is unrestricted.

Note that the `subprocess` module still works when exec is disallowed.

int **allow_threads**

If this is `0` then the sub-interpreter's `threading` module won't create threads. Otherwise threads are allowed.

int **allow_daemon_threads**

If this is `0` then the sub-interpreter's `threading` module won't create daemon threads. Otherwise daemon threads are allowed (as long as `allow_threads` is non-zero).

int **check_multi_interp_extensions**

If this is `0` then all extension modules may be imported, including legacy (single-phase init) modules, in any thread where the sub-interpreter is currently active. Otherwise only multi-phase init extension modules (see **PEP 489**) may be imported. (Also see `Py_mod_multiple_interpreters`.)

This must be `1` (non-zero) if `use_main_obmalloc` is `0`.

int **gil**

This determines the operation of the GIL for the sub-interpreter. It may be one of the following:

PyInterpreterConfig_DEFAULT_GIL

Use the default selection (`PyInterpreterConfig_SHARED_GIL`).

PyInterpreterConfig_SHARED_GIL

Use (share) the main interpreter's GIL.

PyInterpreterConfig_OWN_GIL

Use the sub-interpreter's own GIL.

If this is `PyInterpreterConfig_OWN_GIL` then `PyInterpreterConfig.use_main_obmalloc` must be `0`.

PyStatus **Py_NewInterpreterFromConfig**(*PyThreadState* **tstate_p, const *PyInterpreterConfig* *config)

Create a new sub-interpreter. This is an (almost) totally separate environment for the execution of Python code. In particular, the new interpreter has separate, independent versions of all imported modules, including the fundamental modules `builtins`, `__main__` and `sys`. The table of loaded modules (`sys.modules`) and the module search path (`sys.path`) are also separate. The new environment has no `sys.argv` variable. It has new standard I/O stream file objects `sys.stdin`, `sys.stdout` and `sys.stderr` (however these refer to the same underlying file descriptors).

The given *config* controls the options with which the interpreter is initialized.

Upon success, *tstate_p* will be set to the first thread state created in the new sub-interpreter. This thread state is made in the current thread state. Note that no actual thread is created; see the discussion of thread states below. If creation of the new interpreter is unsuccessful, *tstate_p* is set to `NULL`; no exception is set since the exception state is stored in the current thread state and there may not be a current thread state.

Like all other Python/C API functions, the global interpreter lock must be held before calling this function and is still held when it returns. Likewise a current thread state must be set on entry. On success, the returned thread state will be set as current. If the sub-interpreter is created with its own GIL then the GIL of the calling interpreter will be released. When the function returns, the new interpreter's GIL will be held by the current thread and the previously interpreter's GIL will remain released here.

Added in version 3.12.

Sub-interpreters are most effective when isolated from each other, with certain functionality restricted:

```
PyInterpreterConfig config = {
    .use_main_obmalloc = 0,
    .allow_fork = 0,
    .allow_exec = 0,
    .allow_threads = 1,
    .allow_daemon_threads = 0,
    .check_multi_interp_extensions = 1,
    .gil = PyInterpreterConfig_OWN_GIL,
};
PyThreadState *tstate = Py_NewInterpreterFromConfig(&config);
```

Note that the config is used only briefly and does not get modified. During initialization the config's values are converted into various `PyInterpreterState` values. A read-only copy of the config may be stored internally on the `PyInterpreterState`.

Extension modules are shared between (sub-)interpreters as follows:

- For modules using multi-phase initialization, e.g. `PyModule_FromDefAndSpec()`, a separate module object is created and initialized for each interpreter. Only C-level static and global variables are shared between these module objects.
- For modules using single-phase initialization, e.g. `PyModule_Create()`, the first time a particular extension is imported, it is initialized normally, and a (shallow) copy of its module's dictionary is squirreled away. When the same extension is imported by another (sub-)interpreter, a new module is initialized and filled with the contents of this copy; the extension's `init` function is not called. Objects in the module's dictionary thus end up shared across (sub-)interpreters, which might cause unwanted behavior (see *Bugs and caveats* below).

  Note that this is different from what happens when an extension is imported after the interpreter has been completely re-initialized by calling `Py_FinalizeEx()` and `Py_Initialize()`; in that case, the extension's `initmodule` function *is* called again. As with multi-phase initialization, this means that only C-level static and global variables are shared between these modules.

PyThreadState ***Py_NewInterpreter**(void)

Część stabilnego ABI. Create a new sub-interpreter. This is essentially just a wrapper around `Py_NewInterpreterFromConfig()` with a config that preserves the existing behavior. The result is an unisolated sub-interpreter that shares the main interpreter's GIL, allows fork/exec, allows daemon threads, and allows single-phase init modules.

void **Py_EndInterpreter**(*PyThreadState* *tstate)

Część stabilnego ABI. Destroy the (sub-)interpreter represented by the given thread state. The given thread state must be the current thread state. See the discussion of thread states below. When the call returns, the current thread state is `NULL`. All thread states associated with this interpreter are destroyed. The global interpreter lock used by the target interpreter must be held before calling this function. No GIL is held when it returns.

`Py_FinalizeEx()` will destroy all sub-interpreters that haven't been explicitly destroyed at that point.

9.6.1 A Per-Interpreter GIL

Using `Py_NewInterpreterFromConfig()` you can create a sub-interpreter that is completely isolated from other interpreters, including having its own GIL. The most important benefit of this isolation is that such an interpreter can execute Python code without being blocked by other interpreters or blocking any others. Thus a single Python process can truly take advantage of multiple CPU cores when running Python code. The isolation also encourages a different approach to concurrency than that of just using threads. (See **PEP 554**.)

Using an isolated interpreter requires vigilance in preserving that isolation. That especially means not sharing any objects or mutable state without guarantees about thread-safety. Even objects that are otherwise immutable (e.g. `None`, `(1, 5)`) can't normally be shared because of the refcount. One simple but less-efficient approach around this is to use a global lock around all use of some state (or object). Alternately, effectively immutable objects (like integers or strings) can be made safe in spite of their refcounts by making them *immortal*. In fact, this has been done for the builtin singletons, small integers, and a number of other builtin objects.

If you preserve isolation then you will have access to proper multi-core computing without the complications that come with free-threading. Failure to preserve isolation will expose you to the full consequences of free-threading, including races and hard-to-debug crashes.

Aside from that, one of the main challenges of using multiple isolated interpreters is how to communicate between them safely (not break isolation) and efficiently. The runtime and stdlib do not provide any standard approach to this yet. A future stdlib module would help mitigate the effort of preserving isolation and expose effective tools for communicating (and sharing) data between interpreters.

Added in version 3.12.

9.6.2 Bugs and caveats

Because sub-interpreters (and the main interpreter) are part of the same process, the insulation between them isn't perfect — for example, using low-level file operations like `os.close()` they can (accidentally or maliciously) affect each other's open files. Because of the way extensions are shared between (sub-)interpreters, some extensions may not work properly; this is especially likely when using single-phase initialization or (static) global variables. It is possible to insert objects created in one sub-interpreter into a namespace of another (sub-)interpreter; this should be avoided if possible.

Special care should be taken to avoid sharing user-defined functions, methods, instances or classes between sub--interpreters, since import operations executed by such objects may affect the wrong (sub-)interpreter's dictionary of loaded modules. It is equally important to avoid sharing objects from which the above are reachable.

Also note that combining this functionality with `PyGILState_*` APIs is delicate, because these APIs assume a bijection between Python thread states and OS-level threads, an assumption broken by the presence of sub-interpreters. It is highly recommended that you don't switch sub-interpreters between a pair of matching `PyGILState_Ensure()` and `PyGILState_Release()` calls. Furthermore, extensions (such as `ctypes`) using these APIs to allow calling of Python code from non-Python created threads will probably be broken when using sub-interpreters.

9.7 Asynchronous Notifications

A mechanism is provided to make asynchronous notifications to the main interpreter thread. These notifications take the form of a function pointer and a void pointer argument.

int **Py_AddPendingCall** (int (*func)(void*), void *arg)

Część stabilnego ABI. Schedule a function to be called from the main interpreter thread. On success, `0` is returned and *func* is queued for being called in the main thread. On failure, `-1` is returned without setting any exception.

When successfully queued, *func* will be *eventually* called from the main interpreter thread with the argument *arg*. It will be called asynchronously with respect to normally running Python code, but with both these conditions met:

- on a *bytecode* boundary;
- with the main thread holding the *global interpreter lock* (*func* can therefore use the full C API).

func must return `0` on success, or `-1` on failure with an exception set. *func* won't be interrupted to perform another asynchronous notification recursively, but it can still be interrupted to switch threads if the global interpreter lock is released.

This function doesn't need a current thread state to run, and it doesn't need the global interpreter lock.

To call this function in a subinterpreter, the caller must hold the GIL. Otherwise, the function *func* can be scheduled to be called from the wrong interpreter.

> **Ostrzeżenie**
>
> This is a low-level function, only useful for very special cases. There is no guarantee that *func* will be called as quick as possible. If the main thread is busy executing a system call, *func* won't be called before the system call returns. This function is generally **not** suitable for calling Python code from arbitrary C threads. Instead, use the *PyGILState API*.

Added in version 3.1.

Zmienione w wersji 3.9: If this function is called in a subinterpreter, the function *func* is now scheduled to be called from the subinterpreter, rather than being called from the main interpreter. Each subinterpreter now has its own list of scheduled calls.

9.8 Profiling and Tracing

The Python interpreter provides some low-level support for attaching profiling and execution tracing facilities. These are used for profiling, debugging, and coverage analysis tools.

This C interface allows the profiling or tracing code to avoid the overhead of calling through Python-level callable objects, making a direct C function call instead. The essential attributes of the facility have not changed; the interface allows trace functions to be installed per-thread, and the basic events reported to the trace function are the same as had been reported to the Python-level trace functions in previous versions.

typedef int (***Py_tracefunc**)(*PyObject* *obj, *PyFrameObject* *frame, int what, *PyObject* *arg)

The type of the trace function registered using `PyEval_SetProfile()` and `PyEval_SetTrace()`. The first parameter is the object passed to the registration function as *obj*, *frame* is the frame object to which the event pertains, *what* is one of the constants `PyTrace_CALL`, `PyTrace_EXCEPTION`, `PyTrace_LINE`, `PyTrace_RETURN`, `PyTrace_C_CALL`, `PyTrace_C_EXCEPTION`, `PyTrace_C_RETURN`, or `PyTrace_OPCODE`, and *arg* depends on the value of *what*:

Value of *what*	Meaning of *arg*
`PyTrace_CALL`	Always `Py_None`.
`PyTrace_EXCEPTION`	Exception information as returned by `sys.exc_info()`.
`PyTrace_LINE`	Always `Py_None`.
`PyTrace_RETURN`	Value being returned to the caller, or `NULL` if caused by an exception.
`PyTrace_C_CALL`	Function object being called.
`PyTrace_C_EXCEPTION`	Function object being called.
`PyTrace_C_RETURN`	Function object being called.
`PyTrace_OPCODE`	Always `Py_None`.

int **PyTrace_CALL**

The value of the *what* parameter to a `Py_tracefunc` function when a new call to a function or method is being reported, or a new entry into a generator. Note that the creation of the iterator for a generator function is not reported as there is no control transfer to the Python bytecode in the corresponding frame.

int **PyTrace_EXCEPTION**

The value of the *what* parameter to a `Py_tracefunc` function when an exception has been raised. The callback function is called with this value for *what* when after any bytecode is processed after which the exception becomes set within the frame being executed. The effect of this is that as exception propagation causes the Python stack to unwind, the callback is called upon return to each frame as the exception propagates. Only trace functions receives these events; they are not needed by the profiler.

int **PyTrace_LINE**

The value passed as the *what* parameter to a `Py_tracefunc` function (but not a profiling function) when a line-number event is being reported. It may be disabled for a frame by setting `f_trace_lines` to *0* on that frame.

int **PyTrace_RETURN**

The value for the *what* parameter to `Py_tracefunc` functions when a call is about to return.

int **PyTrace_C_CALL**

The value for the *what* parameter to `Py_tracefunc` functions when a C function is about to be called.

int **PyTrace_C_EXCEPTION**

The value for the *what* parameter to `Py_tracefunc` functions when a C function has raised an exception.

int **PyTrace_C_RETURN**

The value for the *what* parameter to `Py_tracefunc` functions when a C function has returned.

int **PyTrace_OPCODE**

The value for the *what* parameter to `Py_tracefunc` functions (but not profiling functions) when a new opcode is about to be executed. This event is not emitted by default: it must be explicitly requested by setting `f_trace_opcodes` to *1* on the frame.

void **PyEval_SetProfile**(*Py_tracefunc* func, *PyObject* *obj)

Set the profiler function to *func*. The *obj* parameter is passed to the function as its first parameter, and may be any Python object, or `NULL`. If the profile function needs to maintain state, using a different value for *obj* for each thread provides a convenient and thread-safe place to store it. The profile function is called for all monitored events except `PyTrace_LINE` `PyTrace_OPCODE` and `PyTrace_EXCEPTION`.

See also the `sys.setprofile()` function.

The caller must hold the *GIL*.

void **PyEval_SetProfileAllThreads**(*Py_tracefunc* func, *PyObject* *obj)

Like `PyEval_SetProfile()` but sets the profile function in all running threads belonging to the current interpreter instead of the setting it only on the current thread.

The caller must hold the *GIL*.

As `PyEval_SetProfile()`, this function ignores any exceptions raised while setting the profile functions in all threads.

Added in version 3.12.

void **PyEval_SetTrace**(*Py_tracefunc* func, *PyObject* *obj)

Set the tracing function to *func*. This is similar to `PyEval_SetProfile()`, except the tracing function does receive line-number events and per-opcode events, but does not receive any event related to C function objects being called. Any trace function registered using `PyEval_SetTrace()` will not receive `PyTrace_C_CALL`, `PyTrace_C_EXCEPTION` or `PyTrace_C_RETURN` as a value for the *what* parameter.

See also the `sys.settrace()` function.

The caller must hold the *GIL*.

void **PyEval_SetTraceAllThreads**(*Py_tracefunc* func, *PyObject* *obj)

Like `PyEval_SetTrace()` but sets the tracing function in all running threads belonging to the current interpreter instead of the setting it only on the current thread.

The caller must hold the *GIL*.

As `PyEval_SetTrace()`, this function ignores any exceptions raised while setting the trace functions in all threads.

Added in version 3.12.

9.9 Reference tracing

Added in version 3.13.

typedef int (***PyRefTracer**)(*PyObject**, int event, void *data)

The type of the trace function registered using `PyRefTracer_SetTracer()`. The first parameter is a Python object that has been just created (when **event** is set to `PyRefTracer_CREATE`) or about to be destroyed (when **event** is set to `PyRefTracer_DESTROY`). The **data** argument is the opaque pointer that was provided when `PyRefTracer_SetTracer()` was called.

Added in version 3.13.

int **PyRefTracer_CREATE**

The value for the *event* parameter to `PyRefTracer` functions when a Python object has been created.

int **PyRefTracer_DESTROY**

The value for the *event* parameter to `PyRefTracer` functions when a Python object has been destroyed.

int **PyRefTracer_SetTracer**(*PyRefTracer* tracer, void *data)

Register a reference tracer function. The function will be called when a new Python has been created or when an object is going to be destroyed. If **data** is provided it must be an opaque pointer that will be provided when the tracer function is called. Return `0` on success. Set an exception and return `-1` on error.

Not that tracer functions **must not** create Python objects inside or otherwise the call will be re-entrant. The tracer also **must not** clear any existing exception or set an exception. The GIL will be held every time the tracer function is called.

The GIL must be held when calling this function.

Added in version 3.13.

PyRefTracer **PyRefTracer_GetTracer**(void **data)

Get the registered reference tracer function and the value of the opaque data pointer that was registered when `PyRefTracer_SetTracer()` was called. If no tracer was registered this function will return NULL and will set the **data** pointer to NULL.

The GIL must be held when calling this function.

Added in version 3.13.

9.10 Advanced Debugger Support

These functions are only intended to be used by advanced debugging tools.

PyInterpreterState ***PyInterpreterState_Head**()

Return the interpreter state object at the head of the list of all such objects.

PyInterpreterState ***PyInterpreterState_Main**()

Return the main interpreter state object.

PyInterpreterState ***PyInterpreterState_Next**(*PyInterpreterState* *interp)

Return the next interpreter state object after *interp* from the list of all such objects.

PyThreadState ***PyInterpreterState_ThreadHead**(*PyInterpreterState* *interp)

Return the pointer to the first `PyThreadState` object in the list of threads associated with the interpreter *interp*.

PyThreadState ***PyThreadState_Next**(*PyThreadState* *tstate)

Return the next thread state object after *tstate* from the list of all such objects belonging to the same `PyInterpreterState` object.

9.11 Thread Local Storage Support

The Python interpreter provides low-level support for thread-local storage (TLS) which wraps the underlying native TLS implementation to support the Python-level thread local storage API (`threading.local`). The CPython C level APIs are similar to those offered by pthreads and Windows: use a thread key and functions to associate a `void*` value per thread.

The GIL does *not* need to be held when calling these functions; they supply their own locking.

Note that `Python.h` does not include the declaration of the TLS APIs, you need to include `pythread.h` to use thread-local storage.

Informacja

None of these API functions handle memory management on behalf of the `void*` values. You need to allocate and deallocate them yourself. If the `void*` values happen to be `PyObject*`, these functions don't do refcount operations on them either.

9.11.1 Thread Specific Storage (TSS) API

TSS API is introduced to supersede the use of the existing TLS API within the CPython interpreter. This API uses a new type `Py_tss_t` instead of `int` to represent thread keys.

Added in version 3.7.

Zobacz także

„A New C-API for Thread-Local Storage in CPython" (**PEP 539**)

type **Py_tss_t**

This data structure represents the state of a thread key, the definition of which may depend on the underlying TLS implementation, and it has an internal field representing the key's initialization state. There are no public members in this structure.

When *Py_LIMITED_API* is not defined, static allocation of this type by `Py_tss_NEEDS_INIT` is allowed.

Py_tss_NEEDS_INIT

This macro expands to the initializer for `Py_tss_t` variables. Note that this macro won't be defined with *Py_LIMITED_API*.

Dynamic Allocation

Dynamic allocation of the `Py_tss_t`, required in extension modules built with *Py_LIMITED_API*, where static allocation of this type is not possible due to its implementation being opaque at build time.

Py_tss_t ***PyThread_tss_alloc**()

Część stabilnego ABI *od wersji 3.7.* Return a value which is the same state as a value initialized with `Py_tss_NEEDS_INIT`, or `NULL` in the case of dynamic allocation failure.

void **PyThread_tss_free**(*Py_tss_t* *key)

Część stabilnego ABI *od wersji 3.7.* Free the given *key* allocated by `PyThread_tss_alloc()`, after first calling `PyThread_tss_delete()` to ensure any associated thread locals have been unassigned. This is a no-op if the *key* argument is `NULL`.

> **Informacja**
>
> A freed key becomes a dangling pointer. You should reset the key to `NULL`.

Methods

The parameter *key* of these functions must not be `NULL`. Moreover, the behaviors of `PyThread_tss_set()` and `PyThread_tss_get()` are undefined if the given `Py_tss_t` has not been initialized by `PyThread_tss_create()`.

int **PyThread_tss_is_created**(*Py_tss_t* *key)

Część stabilnego ABI *od wersji 3.7.* Return a non-zero value if the given `Py_tss_t` has been initialized by `PyThread_tss_create()`.

int **PyThread_tss_create**(*Py_tss_t* *key)

Część stabilnego ABI *od wersji 3.7.* Return a zero value on successful initialization of a TSS key. The behavior is undefined if the value pointed to by the *key* argument is not initialized by `Py_tss_NEEDS_INIT`. This function can be called repeatedly on the same key – calling it on an already initialized key is a no-op and immediately returns success.

void **PyThread_tss_delete**(*Py_tss_t* *key)

Część stabilnego ABI *od wersji 3.7.* Destroy a TSS key to forget the values associated with the key across all threads, and change the key's initialization state to uninitialized. A destroyed key is able to be initialized again by `PyThread_tss_create()`. This function can be called repeatedly on the same key – calling it on an already destroyed key is a no-op.

int **PyThread_tss_set**(*Py_tss_t* *key, void *value)

Część stabilnego ABI *od wersji 3.7.* Return a zero value to indicate successfully associating a `void*` value with a TSS key in the current thread. Each thread has a distinct mapping of the key to a `void*` value.

void ***PyThread_tss_get**(*Py_tss_t* *key)

Część stabilnego ABI *od wersji 3.7.* Return the `void*` value associated with a TSS key in the current thread. This returns `NULL` if no value is associated with the key in the current thread.

9.11.2 Thread Local Storage (TLS) API

Niezalecane od wersji 3.7: This API is superseded by *Thread Specific Storage (TSS) API*.

> **Informacja**
>
> This version of the API does not support platforms where the native TLS key is defined in a way that cannot be safely cast to `int`. On such platforms, *`PyThread_create_key()`* will return immediately with a failure status, and the other TLS functions will all be no-ops on such platforms.

Due to the compatibility problem noted above, this version of the API should not be used in new code.

int **`PyThread_create_key`**()

Część stabilnego ABI.

void **`PyThread_delete_key`**(int key)

Część stabilnego ABI.

int **`PyThread_set_key_value`**(int key, void *value)

Część stabilnego ABI.

void ***`PyThread_get_key_value`**(int key)

Część stabilnego ABI.

void **`PyThread_delete_key_value`**(int key)

Część stabilnego ABI.

void **`PyThread_ReInitTLS`**()

Część stabilnego ABI.

9.12 Synchronization Primitives

The C-API provides a basic mutual exclusion lock.

type **`PyMutex`**

A mutual exclusion lock. The `PyMutex` should be initialized to zero to represent the unlocked state. For example:

```
PyMutex mutex = {0};
```

Instances of `PyMutex` should not be copied or moved. Both the contents and address of a `PyMutex` are meaningful, and it must remain at a fixed, writable location in memory.

> **Informacja**
>
> A `PyMutex` currently occupies one byte, but the size should be considered unstable. The size may change in future Python releases without a deprecation period.

Added in version 3.13.

void **`PyMutex_Lock`**(*PyMutex* *m)

Lock mutex *m*. If another thread has already locked it, the calling thread will block until the mutex is unlocked. While blocked, the thread will temporarily release the *GIL* if it is held.

Added in version 3.13.

void **`PyMutex_Unlock`**(*PyMutex* *m)

Unlock mutex *m*. The mutex must be locked — otherwise, the function will issue a fatal error.

Added in version 3.13.

9.12.1 Python Critical Section API

The critical section API provides a deadlock avoidance layer on top of per-object locks for *free-threaded* CPython. They are intended to replace reliance on the *global interpreter lock*, and are no-ops in versions of Python with the global interpreter lock.

Critical sections avoid deadlocks by implicitly suspending active critical sections and releasing the locks during calls to `PyEval_SaveThread()`. When `PyEval_RestoreThread()` is called, the most recent critical section is resumed, and its locks reacquired. This means the critical section API provides weaker guarantees than traditional locks – they are useful because their behavior is similar to the *GIL*.

The functions and structs used by the macros are exposed for cases where C macros are not available. They should only be used as in the given macro expansions. Note that the sizes and contents of the structures may change in future Python versions.

> **Informacja**
>
> Operations that need to lock two objects at once must use `Py_BEGIN_CRITICAL_SECTION2`. You *cannot* use nested critical sections to lock more than one object at once, because the inner critical section may suspend the outer critical sections. This API does not provide a way to lock more than two objects at once.

Example usage:

```
static PyObject *
set_field(MyObject *self, PyObject *value)
{
   Py_BEGIN_CRITICAL_SECTION(self);
   Py_SETREF(self->field, Py_XNewRef(value));
   Py_END_CRITICAL_SECTION();
   Py_RETURN_NONE;
}
```

In the above example, `Py_SETREF` calls `Py_DECREF`, which can call arbitrary code through an object's deallocation function. The critical section API avoids potentital deadlocks due to reentrancy and lock ordering by allowing the runtime to temporarily suspend the critical section if the code triggered by the finalizer blocks and calls `PyEval_SaveThread()`.

Py_BEGIN_CRITICAL_SECTION(op)

Acquires the per-object lock for the object *op* and begins a critical section.

In the free-threaded build, this macro expands to:

```
{
    PyCriticalSection _py_cs;
    PyCriticalSection_Begin(&_py_cs, (PyObject*)(op))
```

In the default build, this macro expands to `{`.

Added in version 3.13.

Py_END_CRITICAL_SECTION()

Ends the critical section and releases the per-object lock.

In the free-threaded build, this macro expands to:

```
    PyCriticalSection_End(&_py_cs);
}
```

In the default build, this macro expands to `}`.

Added in version 3.13.

Py_BEGIN_CRITICAL_SECTION2(a, b)

Acquires the per-objects locks for the objects *a* and *b* and begins a critical section. The locks are acquired in a consistent order (lowest address first) to avoid lock ordering deadlocks.

In the free-threaded build, this macro expands to:

```
{
    PyCriticalSection2 _py_cs2;
    PyCriticalSection_Begin2(&_py_cs2, (PyObject*)(a), (PyObject*)(b))
```

In the default build, this macro expands to `{`.

Added in version 3.13.

Py_END_CRITICAL_SECTION2()

Ends the critical section and releases the per-object locks.

In the free-threaded build, this macro expands to:

```
    PyCriticalSection_End2(&_py_cs2);
}
```

In the default build, this macro expands to `}`.

Added in version 3.13.

ROZDZIAŁ 10

Python Initialization Configuration

Added in version 3.8.

Python can be initialized with `Py_InitializeFromConfig()` and the `PyConfig` structure. It can be preinitialized with `Py_PreInitialize()` and the `PyPreConfig` structure.

There are two kinds of configuration:

- The *Python Configuration* can be used to build a customized Python which behaves as the regular Python. For example, environment variables and command line arguments are used to configure Python.
- The *Isolated Configuration* can be used to embed Python into an application. It isolates Python from the system. For example, environment variables are ignored, the LC_CTYPE locale is left unchanged and no signal handler is registered.

The `Py_RunMain()` function can be used to write a customized Python program.

See also *Initialization, Finalization, and Threads*.

Zobacz także

PEP 587 „Python Initialization Configuration".

10.1 Example

Example of customized Python always running in isolated mode:

```
int main(int argc, char **argv)
{
    PyStatus status;

    PyConfig config;
    PyConfig_InitPythonConfig(&config);
    config.isolated = 1;

    /* Decode command line arguments.
       Implicitly preinitialize Python (in isolated mode). */
    status = PyConfig_SetBytesArgv(&config, argc, argv);
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
    if (PyStatus_Exception(status)) {
        goto exception;
    }

    status = Py_InitializeFromConfig(&config);
    if (PyStatus_Exception(status)) {
        goto exception;
    }
    PyConfig_Clear(&config);

    return Py_RunMain();

exception:
    PyConfig_Clear(&config);
    if (PyStatus_IsExit(status)) {
        return status.exitcode;
    }
    /* Display the error message and exit the process with
       non-zero exit code */
    Py_ExitStatusException(status);
}
```

10.2 PyWideStringList

type **PyWideStringList**

List of `wchar_t*` strings.

If *length* is non-zero, *items* must be non-`NULL` and all strings must be non-`NULL`.

Methods:

PyStatus **PyWideStringList_Append**(*PyWideStringList* *list, const wchar_t *item)

Append *item* to *list*.

Python must be preinitialized to call this function.

PyStatus **PyWideStringList_Insert**(*PyWideStringList* *list, *Py_ssize_t* index, const wchar_t *item)

Insert *item* into *list* at *index*.

If *index* is greater than or equal to *list* length, append *item* to *list*.

index must be greater than or equal to `0`.

Python must be preinitialized to call this function.

Structure fields:

Py_ssize_t **length**

List length.

wchar_t ****items**

List items.

10.3 PyStatus

type **PyStatus**

Structure to store an initialization function status: success, error or exit.

For an error, it can store the C function name which created the error.

Structure fields:

int **exitcode**

Exit code. Argument passed to `exit()`.

const char ***err_msg**

Error message.

const char ***func**

Name of the function which created an error, can be `NULL`.

Functions to create a status:

PyStatus **PyStatus_Ok**(void)

Success.

PyStatus **PyStatus_Error**(const char *err_msg)

Initialization error with a message.

err_msg must not be `NULL`.

PyStatus **PyStatus_NoMemory**(void)

Memory allocation failure (out of memory).

PyStatus **PyStatus_Exit**(int exitcode)

Exit Python with the specified exit code.

Functions to handle a status:

int **PyStatus_Exception**(*PyStatus* status)

Is the status an error or an exit? If true, the exception must be handled; by calling `Py_ExitStatusException()` for example.

int **PyStatus_IsError**(*PyStatus* status)

Is the result an error?

int **PyStatus_IsExit**(*PyStatus* status)

Is the result an exit?

void **Py_ExitStatusException**(*PyStatus* status)

Call `exit(exitcode)` if *status* is an exit. Print the error message and exit with a non-zero exit code if *status* is an error. Must only be called if `PyStatus_Exception(status)` is non-zero.

Informacja

Internally, Python uses macros which set `PyStatus.func`, whereas functions to create a status set `func` to `NULL`.

Example:

```
PyStatus alloc(void **ptr, size_t size)
{
    *ptr = PyMem_RawMalloc(size);
    if (*ptr == NULL) {
        return PyStatus_NoMemory();
    }
    return PyStatus_Ok();
}

int main(int argc, char **argv)
{
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
    void *ptr;
    PyStatus status = alloc(&ptr, 16);
    if (PyStatus_Exception(status)) {
        Py_ExitStatusException(status);
    }
    PyMem_Free(ptr);
    return 0;
}
```

10.4 PyPreConfig

type **PyPreConfig**

Structure used to preinitialize Python.

Function to initialize a preconfiguration:

void **PyPreConfig_InitPythonConfig**(*PyPreConfig* *preconfig)

Initialize the preconfiguration with *Python Configuration*.

void **PyPreConfig_InitIsolatedConfig**(*PyPreConfig* *preconfig)

Initialize the preconfiguration with *Isolated Configuration*.

Structure fields:

int **allocator**

Name of the Python memory allocators:

- `PYMEM_ALLOCATOR_NOT_SET` (`0`): don't change memory allocators (use defaults).
- `PYMEM_ALLOCATOR_DEFAULT` (`1`): *default memory allocators*.
- `PYMEM_ALLOCATOR_DEBUG` (`2`): *default memory allocators* with *debug hooks*.
- `PYMEM_ALLOCATOR_MALLOC` (`3`): use `malloc()` of the C library.
- `PYMEM_ALLOCATOR_MALLOC_DEBUG` (`4`): force usage of `malloc()` with *debug hooks*.
- `PYMEM_ALLOCATOR_PYMALLOC` (`5`): *Python pymalloc memory allocator*.
- `PYMEM_ALLOCATOR_PYMALLOC_DEBUG` (`6`): *Python pymalloc memory allocator* with *debug hooks*.
- `PYMEM_ALLOCATOR_MIMALLOC` (`6`): use `mimalloc`, a fast malloc replacement.
- `PYMEM_ALLOCATOR_MIMALLOC_DEBUG` (`7`): use `mimalloc`, a fast malloc replacement with *debug hooks*.

`PYMEM_ALLOCATOR_PYMALLOC` and `PYMEM_ALLOCATOR_PYMALLOC_DEBUG` are not supported if Python is `configured using --without-pymalloc`.

`PYMEM_ALLOCATOR_MIMALLOC` and `PYMEM_ALLOCATOR_MIMALLOC_DEBUG` are not supported if Python is `configured using --without-mimalloc` or if the underlying atomic support isn't available.

See *Memory Management*.

Default: `PYMEM_ALLOCATOR_NOT_SET`.

int **configure_locale**

Set the LC_CTYPE locale to the user preferred locale.

If equals to `0`, set `coerce_c_locale` and `coerce_c_locale_warn` members to `0`.

See the *locale encoding*.

Default: `1` in Python config, `0` in isolated config.

int **coerce_c_locale**

If equals to `2`, coerce the C locale.

If equals to `1`, read the LC_CTYPE locale to decide if it should be coerced.

See the *locale encoding*.

Default: `-1` in Python config, `0` in isolated config.

int **coerce_c_locale_warn**

If non-zero, emit a warning if the C locale is coerced.

Default: `-1` in Python config, `0` in isolated config.

int **dev_mode**

Python Development Mode: see `PyConfig.dev_mode`.

Default: `-1` in Python mode, `0` in isolated mode.

int **isolated**

Isolated mode: see `PyConfig.isolated`.

Default: `0` in Python mode, `1` in isolated mode.

int **legacy_windows_fs_encoding**

If non-zero:

- Set `PyPreConfig.utf8_mode` to `0`,
- Set `PyConfig.filesystem_encoding` to `"mbcs"`,
- Set `PyConfig.filesystem_errors` to `"replace"`.

Initialized from the `PYTHONLEGACYWINDOWSFSENCODING` environment variable value.

Only available on Windows. `#ifdef MS_WINDOWS` macro can be used for Windows specific code.

Default: `0`.

int **parse_argv**

If non-zero, `Py_PreInitializeFromArgs()` and `Py_PreInitializeFromBytesArgs()` parse their `argv` argument the same way the regular Python parses command line arguments: see Command Line Arguments.

Default: `1` in Python config, `0` in isolated config.

int **use_environment**

Use environment variables? See `PyConfig.use_environment`.

Default: `1` in Python config and `0` in isolated config.

int **utf8_mode**

If non-zero, enable the Python UTF-8 Mode.

Set to `0` or `1` by the `-X utf8` command line option and the `PYTHONUTF8` environment variable.

Also set to `1` if the `LC_CTYPE` locale is `C` or `POSIX`.

Default: `-1` in Python config and `0` in isolated config.

10.5 Preinitialize Python with PyPreConfig

The preinitialization of Python:

- Set the Python memory allocators (`PyPreConfig.allocator`)
- Configure the LC_CTYPE locale (*locale encoding*)
- Set the Python UTF-8 Mode (`PyPreConfig.utf8_mode`)

The current preconfiguration (`PyPreConfig` type) is stored in `_PyRuntime.preconfig`.

Functions to preinitialize Python:

PyStatus **`Py_PreInitialize`**(const *PyPreConfig* *preconfig)

Preinitialize Python from *preconfig* preconfiguration.

preconfig must not be `NULL`.

PyStatus **`Py_PreInitializeFromBytesArgs`**(const *PyPreConfig* *preconfig, int argc, char *const *argv)

Preinitialize Python from *preconfig* preconfiguration.

Parse *argv* command line arguments (bytes strings) if `parse_argv` of *preconfig* is non-zero.

preconfig must not be `NULL`.

PyStatus **`Py_PreInitializeFromArgs`**(const *PyPreConfig* *preconfig, int argc, wchar_t *const *argv)

Preinitialize Python from *preconfig* preconfiguration.

Parse *argv* command line arguments (wide strings) if `parse_argv` of *preconfig* is non-zero.

preconfig must not be `NULL`.

The caller is responsible to handle exceptions (error or exit) using `PyStatus_Exception()` and `Py_ExitStatusException()`.

For *Python Configuration* (`PyPreConfig_InitPythonConfig()`), if Python is initialized with command line arguments, the command line arguments must also be passed to preinitialize Python, since they have an effect on the pre-configuration like encodings. For example, the `-X utf8` command line option enables the Python UTF-8 Mode.

`PyMem_SetAllocator()` can be called after `Py_PreInitialize()` and before `Py_InitializeFromConfig()` to install a custom memory allocator. It can be called before `Py_PreInitialize()` if `PyPreConfig.allocator` is set to `PYMEM_ALLOCATOR_NOT_SET`.

Python memory allocation functions like `PyMem_RawMalloc()` must not be used before the Python preinitialization, whereas calling directly `malloc()` and `free()` is always safe. `Py_DecodeLocale()` must not be called before the Python preinitialization.

Example using the preinitialization to enable the Python UTF-8 Mode:

```
PyStatus status;
PyPreConfig preconfig;
PyPreConfig_InitPythonConfig(&preconfig);

preconfig.utf8_mode = 1;

status = Py_PreInitialize(&preconfig);
if (PyStatus_Exception(status)) {
    Py_ExitStatusException(status);
}

/* at this point, Python speaks UTF-8 */

Py_Initialize();
/* ... use Python API here ... */
Py_Finalize();
```

10.6 PyConfig

type **`PyConfig`**

Structure containing most parameters to configure Python.

When done, the `PyConfig_Clear()` function must be used to release the configuration memory.

Structure methods:

void **PyConfig_InitPythonConfig**(*PyConfig* *config)

Initialize configuration with the *Python Configuration*.

void **PyConfig_InitIsolatedConfig**(*PyConfig* *config)

Initialize configuration with the *Isolated Configuration*.

PyStatus **PyConfig_SetString**(*PyConfig* *config, wchar_t *const *config_str, const wchar_t *str)

Copy the wide character string *str* into `*config_str`.

Preinitialize Python if needed.

PyStatus **PyConfig_SetBytesString**(*PyConfig* *config, wchar_t *const *config_str, const char *str)

Decode *str* using `Py_DecodeLocale()` and set the result into `*config_str`.

Preinitialize Python if needed.

PyStatus **PyConfig_SetArgv**(*PyConfig* *config, int argc, wchar_t *const *argv)

Set command line arguments (`argv` member of *config*) from the *argv* list of wide character strings.

Preinitialize Python if needed.

PyStatus **PyConfig_SetBytesArgv**(*PyConfig* *config, int argc, char *const *argv)

Set command line arguments (`argv` member of *config*) from the *argv* list of bytes strings. Decode bytes using `Py_DecodeLocale()`.

Preinitialize Python if needed.

PyStatus **PyConfig_SetWideStringList**(*PyConfig* *config, *PyWideStringList* *list, *Py_ssize_t* length, wchar_t **items)

Set the list of wide strings *list* to *length* and *items*.

Preinitialize Python if needed.

PyStatus **PyConfig_Read**(*PyConfig* *config)

Read all Python configuration.

Fields which are already initialized are left unchanged.

Fields for *path configuration* are no longer calculated or modified when calling this function, as of Python 3.11.

The `PyConfig_Read()` function only parses `PyConfig.argv` arguments once: `PyConfig.parse_argv` is set to `2` after arguments are parsed. Since Python arguments are stripped from `PyConfig.argv`, parsing arguments twice would parse the application options as Python options.

Preinitialize Python if needed.

Zmienione w wersji 3.10: The `PyConfig.argv` arguments are now only parsed once, `PyConfig.parse_argv` is set to `2` after arguments are parsed, and arguments are only parsed if `PyConfig.parse_argv` equals `1`.

Zmienione w wersji 3.11: `PyConfig_Read()` no longer calculates all paths, and so fields listed under *Python Path Configuration* may no longer be updated until `Py_InitializeFromConfig()` is called.

void **PyConfig_Clear**(*PyConfig* *config)

Release configuration memory.

Most `PyConfig` methods *preinitialize Python* if needed. In that case, the Python preinitialization configuration (`PyPreConfig`) in based on the `PyConfig`. If configuration fields which are in common with `PyPreConfig` are tuned, they must be set before calling a `PyConfig` method:

- `PyConfig.dev_mode`
- `PyConfig.isolated`
- `PyConfig.parse_argv`

- *PyConfig.use_environment*

Moreover, if *PyConfig_SetArgv()* or *PyConfig_SetBytesArgv()* is used, this method must be called before other methods, since the preinitialization configuration depends on command line arguments (if *parse_argv* is non-zero).

The caller of these methods is responsible to handle exceptions (error or exit) using `PyStatus_Exception()` and `Py_ExitStatusException()`.

Structure fields:

PyWideStringList **argv**

Set `sys.argv` command line arguments based on *argv*. These parameters are similar to those passed to the program's `main()` function with the difference that the first entry should refer to the script file to be executed rather than the executable hosting the Python interpreter. If there isn't a script that will be run, the first entry in *argv* can be an empty string.

Set *parse_argv* to `1` to parse *argv* the same way the regular Python parses Python command line arguments and then to strip Python arguments from *argv*.

If *argv* is empty, an empty string is added to ensure that `sys.argv` always exists and is never empty.

Default: `NULL`.

See also the *orig_argv* member.

int **safe_path**

If equals to zero, `Py_RunMain()` prepends a potentially unsafe path to `sys.path` at startup:

- If *argv[0]* is equal to `L"-m"` (`python -m module`), prepend the current working directory.
- If running a script (`python script.py`), prepend the script's directory. If it's a symbolic link, resolve symbolic links.
- Otherwise (`python -c code` and `python`), prepend an empty string, which means the current working directory.

Set to `1` by the `-P` command line option and the `PYTHONSAFEPATH` environment variable.

Default: `0` in Python config, `1` in isolated config.

Added in version 3.11.

wchar_t ***base_exec_prefix**

`sys.base_exec_prefix`.

Default: `NULL`.

Part of the *Python Path Configuration* output.

See also *PyConfig.exec_prefix*.

wchar_t ***base_executable**

Python base executable: `sys._base_executable`.

Set by the `__PYVENV_LAUNCHER__` environment variable.

Set from *PyConfig.executable* if `NULL`.

Default: `NULL`.

Part of the *Python Path Configuration* output.

See also *PyConfig.executable*.

wchar_t ***base_prefix**

`sys.base_prefix`.

Default: `NULL`.

Part of the *Python Path Configuration* output.

See also *PyConfig.prefix*.

int **buffered_stdio**

If equals to `0` and *configure_c_stdio* is non-zero, disable buffering on the C streams stdout and stderr.

Set to `0` by the `-u` command line option and the `PYTHONUNBUFFERED` environment variable.

stdin is always opened in buffered mode.

Default: `1`.

int **bytes_warning**

If equals to `1`, issue a warning when comparing `bytes` or `bytearray` with `str`, or comparing `bytes` with `int`.

If equal or greater to `2`, raise a `BytesWarning` exception in these cases.

Incremented by the `-b` command line option.

Default: `0`.

int **warn_default_encoding**

If non-zero, emit a `EncodingWarning` warning when `io.TextIOWrapper` uses its default encoding. See io-encoding-warning for details.

Default: `0`.

Added in version 3.10.

int **code_debug_ranges**

If equals to `0`, disables the inclusion of the end line and column mappings in code objects. Also disables traceback printing carets to specific error locations.

Set to `0` by the `PYTHONNODEBUGRANGES` environment variable and by the `-X no_debug_ranges` command line option.

Default: `1`.

Added in version 3.11.

wchar_t ***check_hash_pycs_mode**

Control the validation behavior of hash-based `.pyc` files: value of the `--check-hash-based-pycs` command line option.

Valid values:

- `L"always"`: Hash the source file for invalidation regardless of value of the «check_source» flag.
- `L"never"`: Assume that hash-based pycs always are valid.
- `L"default"`: The «check_source» flag in hash-based pycs determines invalidation.

Default: `L"default"`.

See also **PEP 552** „Deterministic pycs”.

int **configure_c_stdio**

If non-zero, configure C standard streams:

- On Windows, set the binary mode (`O_BINARY`) on stdin, stdout and stderr.
- If *buffered_stdio* equals zero, disable buffering of stdin, stdout and stderr streams.
- If *interactive* is non-zero, enable stream buffering on stdin and stdout (only stdout on Windows).

Default: `1` in Python config, `0` in isolated config.

int **dev_mode**

If non-zero, enable the Python Development Mode.

Set to `1` by the `-X dev` option and the `PYTHONDEVMODE` environment variable.

Default: `-1` in Python mode, `0` in isolated mode.

int **dump_refs**

Dump Python references?

If non-zero, dump all objects which are still alive at exit.

Set to `1` by the `PYTHONDUMPREFS` environment variable.

Needs a special build of Python with the `Py_TRACE_REFS` macro defined: see the `configure --with-trace-refs option`.

Default: `0`.

wchar_t ***exec_prefix**

The site-specific directory prefix where the platform-dependent Python files are installed: `sys.exec_prefix`.

Default: `NULL`.

Part of the *Python Path Configuration* output.

See also `PyConfig.base_exec_prefix`.

wchar_t ***executable**

The absolute path of the executable binary for the Python interpreter: `sys.executable`.

Default: `NULL`.

Part of the *Python Path Configuration* output.

See also `PyConfig.base_executable`.

int **faulthandler**

Enable faulthandler?

If non-zero, call `faulthandler.enable()` at startup.

Set to `1` by `-X faulthandler` and the `PYTHONFAULTHANDLER` environment variable.

Default: `-1` in Python mode, `0` in isolated mode.

wchar_t ***filesystem_encoding**

Filesystem encoding: `sys.getfilesystemencoding()`.

On macOS, Android and VxWorks: use `"utf-8"` by default.

On Windows: use `"utf-8"` by default, or `"mbcs"` if `legacy_windows_fs_encoding` of `PyPreConfig` is non-zero.

Default encoding on other platforms:

- `"utf-8"` if `PyPreConfig.utf8_mode` is non-zero.
- `"ascii"` if Python detects that `nl_langinfo(CODESET)` announces the ASCII encoding, whereas the `mbstowcs()` function decodes from a different encoding (usually Latin1).
- `"utf-8"` if `nl_langinfo(CODESET)` returns an empty string.
- Otherwise, use the *locale encoding*: `nl_langinfo(CODESET)` result.

At Python startup, the encoding name is normalized to the Python codec name. For example, `"ANSI_X3.4-1968"` is replaced with `"ascii"`.

See also the `filesystem_errors` member.

wchar_t ***filesystem_errors**

Filesystem error handler: `sys.getfilesystemencodeerrors()`.

On Windows: use `"surrogatepass"` by default, or `"replace"` if `legacy_windows_fs_encoding` of `PyPreConfig` is non-zero.

On other platforms: use `"surrogateescape"` by default.

Supported error handlers:

- `"strict"`
- `"surrogateescape"`
- `"surrogatepass"` (only supported with the UTF-8 encoding)

See also the `filesystem_encoding` member.

unsigned long **hash_seed**

int **use_hash_seed**

Randomized hash function seed.

If `use_hash_seed` is zero, a seed is chosen randomly at Python startup, and `hash_seed` is ignored.

Set by the `PYTHONHASHSEED` environment variable.

Default *use_hash_seed* value: `-1` in Python mode, `0` in isolated mode.

wchar_t ***home**

Set the default Python „home" directory, that is, the location of the standard Python libraries (see `PYTHONHOME`).

Set by the `PYTHONHOME` environment variable.

Default: `NULL`.

Part of the *Python Path Configuration* input.

int **import_time**

If non-zero, profile import time.

Set the `1` by the `-X importtime` option and the `PYTHONPROFILEIMPORTTIME` environment variable.

Default: `0`.

int **inspect**

Enter interactive mode after executing a script or a command.

If greater than `0`, enable inspect: when a script is passed as first argument or the -c option is used, enter interactive mode after executing the script or the command, even when `sys.stdin` does not appear to be a terminal.

Incremented by the `-i` command line option. Set to `1` if the `PYTHONINSPECT` environment variable is non-empty.

Default: `0`.

int **install_signal_handlers**

Install Python signal handlers?

Default: `1` in Python mode, `0` in isolated mode.

int **interactive**

If greater than `0`, enable the interactive mode (REPL).

Incremented by the `-i` command line option.

Default: `0`.

int **int_max_str_digits**

Configures the integer string conversion length limitation. An initial value of `-1` means the value will be taken from the command line or environment or otherwise default to 4300 (`sys.int_info.default_max_str_digits`). A value of `0` disables the limitation. Values greater than zero but less than 640 (`sys.int_info.str_digits_check_threshold`) are unsupported and will produce an error.

Configured by the `-X int_max_str_digits` command line flag or the `PYTHONINTMAXSTRDIGITS` environment variable.

Default: `-1` in Python mode. 4300 (`sys.int_info.default_max_str_digits`) in isolated mode.

Added in version 3.12.

int **cpu_count**

If the value of *cpu_count* is not `-1` then it will override the return values of `os.cpu_count()`, `os.process_cpu_count()`, and `multiprocessing.cpu_count()`.

Configured by the `-X cpu_count=n|default` command line flag or the `PYTHON_CPU_COUNT` environment variable.

Default: `-1`.

Added in version 3.13.

int **isolated**

If greater than `0`, enable isolated mode:

- Set *safe_path* to `1`: don't prepend a potentially unsafe path to `sys.path` at Python startup, such as the current directory, the script's directory or an empty string.
- Set *use_environment* to `0`: ignore `PYTHON` environment variables.
- Set *user_site_directory* to `0`: don't add the user site directory to `sys.path`.
- Python REPL doesn't import `readline` nor enable default readline configuration on interactive prompts.

Set to `1` by the `-I` command line option.

Default: `0` in Python mode, `1` in isolated mode.

See also the *Isolated Configuration* and *PyPreConfig.isolated*.

int **legacy_windows_stdio**

If non-zero, use `io.FileIO` instead of `io._WindowsConsoleIO` for `sys.stdin`, `sys.stdout` and `sys.stderr`.

Set to `1` if the `PYTHONLEGACYWINDOWSSTDIO` environment variable is set to a non-empty string.

Only available on Windows. `#ifdef MS_WINDOWS` macro can be used for Windows specific code.

Default: `0`.

See also the **PEP 528** (Change Windows console encoding to UTF-8).

int **malloc_stats**

If non-zero, dump statistics on *Python pymalloc memory allocator* at exit.

Set to `1` by the `PYTHONMALLOCSTATS` environment variable.

The option is ignored if Python is `configured using the --without-pymalloc option`.

Default: `0`.

wchar_t ***platlibdir**

Platform library directory name: `sys.platlibdir`.

Set by the `PYTHONPLATLIBDIR` environment variable.

Default: value of the `PLATLIBDIR` macro which is set by the `configure --with-platlibdir option` (default: `"lib"`, or `"DLLs"` on Windows).

Part of the *Python Path Configuration* input.

Added in version 3.9.

Zmienione w wersji 3.11: This macro is now used on Windows to locate the standard library extension modules, typically under `DLLs`. However, for compatibility, note that this value is ignored for any non--standard layouts, including in-tree builds and virtual environments.

wchar_t ***pythonpath_env**

Module search paths (`sys.path`) as a string separated by `DELIM` (`os.pathsep`).

Set by the `PYTHONPATH` environment variable.

Default: `NULL`.

Part of the *Python Path Configuration* input.

PyWideStringList **module_search_paths**

int **module_search_paths_set**

Module search paths: `sys.path`.

If `module_search_paths_set` is equal to `0`, `Py_InitializeFromConfig()` will replace `module_search_paths` and sets `module_search_paths_set` to `1`.

Default: empty list (`module_search_paths`) and `0` (`module_search_paths_set`).

Part of the *Python Path Configuration* output.

int **optimization_level**

Compilation optimization level:

- `0`: Peephole optimizer, set `__debug__` to `True`.
- `1`: Level 0, remove assertions, set `__debug__` to `False`.
- `2`: Level 1, strip docstrings.

Incremented by the `-O` command line option. Set to the `PYTHONOPTIMIZE` environment variable value.

Default: `0`.

PyWideStringList **orig_argv**

The list of the original command line arguments passed to the Python executable: `sys.orig_argv`.

If `orig_argv` list is empty and `argv` is not a list only containing an empty string, `PyConfig_Read()` copies `argv` into `orig_argv` before modifying `argv` (if `parse_argv` is non-zero).

See also the `argv` member and the `Py_GetArgcArgv()` function.

Default: empty list.

Added in version 3.10.

int **parse_argv**

Parse command line arguments?

If equals to `1`, parse `argv` the same way the regular Python parses command line arguments, and strip Python arguments from `argv`.

The `PyConfig_Read()` function only parses `PyConfig.argv` arguments once: `PyConfig.parse_argv` is set to `2` after arguments are parsed. Since Python arguments are stripped from `PyConfig.argv`, parsing arguments twice would parse the application options as Python options.

Default: `1` in Python mode, `0` in isolated mode.

Zmienione w wersji 3.10: The `PyConfig.argv` arguments are now only parsed if `PyConfig.parse_argv` equals to `1`.

int **parser_debug**

Parser debug mode. If greater than `0`, turn on parser debugging output (for expert only, depending on compilation options).

Incremented by the `-d` command line option. Set to the `PYTHONDEBUG` environment variable value.

Needs a debug build of Python (the `Py_DEBUG` macro must be defined).

Default: `0`.

int **pathconfig_warnings**

If non-zero, calculation of path configuration is allowed to log warnings into `stderr`. If equals to `0`, suppress these warnings.

Default: `1` in Python mode, `0` in isolated mode.

Part of the *Python Path Configuration* input.

Zmienione w wersji 3.11: Now also applies on Windows.

wchar_t ***prefix**

The site-specific directory prefix where the platform independent Python files are installed: `sys.prefix`.

Default: `NULL`.

Part of the *Python Path Configuration* output.

See also `PyConfig.base_prefix`.

wchar_t ***program_name**

Program name used to initialize `executable` and in early error messages during Python initialization.

- On macOS, use `PYTHONEXECUTABLE` environment variable if set.
- If the `WITH_NEXT_FRAMEWORK` macro is defined, use `__PYVENV_LAUNCHER__` environment variable if set.
- Use `argv[0]` of `argv` if available and non-empty.
- Otherwise, use `L"python"` on Windows, or `L"python3"` on other platforms.

Default: `NULL`.

Part of the *Python Path Configuration* input.

wchar_t ***pycache_prefix**

Directory where cached `.pyc` files are written: `sys.pycache_prefix`.

Set by the `-X pycache_prefix=PATH` command line option and the `PYTHONPYCACHEPREFIX` environment variable. The command-line option takes precedence.

If `NULL`, `sys.pycache_prefix` is set to `None`.

Default: `NULL`.

int **quiet**

Quiet mode. If greater than `0`, don't display the copyright and version at Python startup in interactive mode.

Incremented by the `-q` command line option.

Default: `0`.

wchar_t ***run_command**

Value of the `-c` command line option.

Used by `Py_RunMain()`.

Default: `NULL`.

wchar_t ***run_filename***

Filename passed on the command line: trailing command line argument without `-c` or `-m`. It is used by the *Py_RunMain()* function.

For example, it is set to `script.py` by the `python3 script.py arg` command line.

See also the *PyConfig.skip_source_first_line* option.

Default: `NULL`.

wchar_t ***run_module***

Value of the `-m` command line option.

Used by *Py_RunMain()*.

Default: `NULL`.

wchar_t ***run_presite***

`package.module` path to module that should be imported before `site.py` is run.

Set by the `-X presite=package.module` command-line option and the `PYTHON_PRESITE` environment variable. The command-line option takes precedence.

Needs a debug build of Python (the `Py_DEBUG` macro must be defined).

Default: `NULL`.

int ***show_ref_count***

Show total reference count at exit (excluding *immortal* objects)?

Set to `1` by `-X showrefcount` command line option.

Needs a debug build of Python (the `Py_REF_DEBUG` macro must be defined).

Default: `0`.

int ***site_import***

Import the `site` module at startup?

If equal to zero, disable the import of the module site and the site-dependent manipulations of `sys.path` that it entails.

Also disable these manipulations if the `site` module is explicitly imported later (call `site.main()` if you want them to be triggered).

Set to `0` by the `-S` command line option.

`sys.flags.no_site` is set to the inverted value of *site_import*.

Default: `1`.

int ***skip_source_first_line***

If non-zero, skip the first line of the *PyConfig.run_filename* source.

It allows the usage of non-Unix forms of `#!cmd`. This is intended for a DOS specific hack only.

Set to `1` by the `-x` command line option.

Default: `0`.

wchar_t ***stdio_encoding***

wchar_t ***stdio_errors***

Encoding and encoding errors of `sys.stdin`, `sys.stdout` and `sys.stderr` (but `sys.stderr` always uses `"backslashreplace"` error handler).

Use the `PYTHONIOENCODING` environment variable if it is non-empty.

Default encoding:

- `"UTF-8"` if *PyPreConfig.utf8_mode* is non-zero.

- Otherwise, use the *locale encoding*.

Default error handler:

- On Windows: use `"surrogateescape"`.
- `"surrogateescape"` if `PyPreConfig.utf8_mode` is non-zero, or if the LC_CTYPE locale is „C" or „POSIX".
- `"strict"` otherwise.

See also `PyConfig.legacy_windows_stdio`.

int **tracemalloc**

Enable tracemalloc?

If non-zero, call `tracemalloc.start()` at startup.

Set by `-X tracemalloc=N` command line option and by the `PYTHONTRACEMALLOC` environment variable.

Default: `-1` in Python mode, `0` in isolated mode.

int **perf_profiling**

Enable compatibility mode with the perf profiler?

If non-zero, initialize the perf trampoline. See perf_profiling for more information.

Set by `-X perf` command-line option and by the `PYTHON_PERF_JIT_SUPPORT` environment variable for perf support with stack pointers and `-X perf_jit` command-line option and by the `PYTHON_PERF_JIT_SUPPORT` environment variable for perf support with DWARF JIT information.

Default: `-1`.

Added in version 3.12.

int **use_environment**

Use environment variables?

If equals to zero, ignore the environment variables.

Set to `0` by the `-E` environment variable.

Default: `1` in Python config and `0` in isolated config.

int **user_site_directory**

If non-zero, add the user site directory to `sys.path`.

Set to `0` by the `-s` and `-I` command line options.

Set to `0` by the `PYTHONNOUSERSITE` environment variable.

Default: `1` in Python mode, `0` in isolated mode.

int **verbose**

Verbose mode. If greater than `0`, print a message each time a module is imported, showing the place (filename or built-in module) from which it is loaded.

If greater than or equal to `2`, print a message for each file that is checked for when searching for a module. Also provides information on module cleanup at exit.

Incremented by the `-v` command line option.

Set by the `PYTHONVERBOSE` environment variable value.

Default: `0`.

PyWideStringList **warnoptions**

Options of the `warnings` module to build warnings filters, lowest to highest priority: `sys.warnoptions`.

The `warnings` module adds `sys.warnoptions` in the reverse order: the last *PyConfig.warnoptions* item becomes the first item of `warnings.filters` which is checked first (highest priority).

The `-W` command line options adds its value to *warnoptions*, it can be used multiple times.

The `PYTHONWARNINGS` environment variable can also be used to add warning options. Multiple options can be specified, separated by commas (`,`).

Default: empty list.

int **write_bytecode**

If equal to `0`, Python won't try to write `.pyc` files on the import of source modules.

Set to `0` by the `-B` command line option and the `PYTHONDONTWRITEBYTECODE` environment variable.

`sys.dont_write_bytecode` is initialized to the inverted value of *write_bytecode*.

Default: `1`.

PyWideStringList **xoptions**

Values of the `-X` command line options: `sys._xoptions`.

Default: empty list.

If *parse_argv* is non-zero, *argv* arguments are parsed the same way the regular Python parses command line arguments, and Python arguments are stripped from *argv*.

The *xoptions* options are parsed to set other options: see the `-X` command line option.

Zmienione w wersji 3.9: The `show_alloc_count` field has been removed.

10.7 Initialization with PyConfig

Initializing the interpreter from a populated configuration struct is handled by calling *Py_InitializeFromConfig()*.

The caller is responsible to handle exceptions (error or exit) using *PyStatus_Exception()* and *Py_ExitStatusException()*.

If *PyImport_FrozenModules()*, *PyImport_AppendInittab()* or *PyImport_ExtendInittab()* are used, they must be set or called after Python preinitialization and before the Python initialization. If Python is initialized multiple times, *PyImport_AppendInittab()* or *PyImport_ExtendInittab()* must be called before each Python initialization.

The current configuration (`PyConfig` type) is stored in `PyInterpreterState.config`.

Example setting the program name:

```
void init_python(void)
{
    PyStatus status;

    PyConfig config;
    PyConfig_InitPythonConfig(&config);

    /* Set the program name. Implicitly preinitialize Python. */
    status = PyConfig_SetString(&config, &config.program_name,
                                L"/path/to/my_program");
    if (PyStatus_Exception(status)) {
        goto exception;
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
    }

    status = Py_InitializeFromConfig(&config);
    if (PyStatus_Exception(status)) {
        goto exception;
    }
    PyConfig_Clear(&config);
    return;

exception:
    PyConfig_Clear(&config);
    Py_ExitStatusException(status);
}
```

More complete example modifying the default configuration, read the configuration, and then override some parameters. Note that since 3.11, many parameters are not calculated until initialization, and so values cannot be read from the configuration structure. Any values set before initialize is called will be left unchanged by initialization:

```
PyStatus init_python(const char *program_name)
{
    PyStatus status;

    PyConfig config;
    PyConfig_InitPythonConfig(&config);

    /* Set the program name before reading the configuration
       (decode byte string from the locale encoding).

       Implicitly preinitialize Python. */
    status = PyConfig_SetBytesString(&config, &config.program_name,
                                     program_name);
    if (PyStatus_Exception(status)) {
        goto done;
    }

    /* Read all configuration at once */
    status = PyConfig_Read(&config);
    if (PyStatus_Exception(status)) {
        goto done;
    }

    /* Specify sys.path explicitly */
    /* If you want to modify the default set of paths, finish
       initialization first and then use PySys_GetObject("path") */
    config.module_search_paths_set = 1;
    status = PyWideStringList_Append(&config.module_search_paths,
                                     L"/path/to/stdlib");
    if (PyStatus_Exception(status)) {
        goto done;
    }
    status = PyWideStringList_Append(&config.module_search_paths,
                                     L"/path/to/more/modules");
    if (PyStatus_Exception(status)) {
        goto done;
    }
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
    /* Override executable computed by PyConfig_Read() */
    status = PyConfig_SetString(&config, &config.executable,
                                L"/path/to/my_executable");
    if (PyStatus_Exception(status)) {
        goto done;
    }

    status = Py_InitializeFromConfig(&config);

done:
    PyConfig_Clear(&config);
    return status;
}
```

10.8 Isolated Configuration

`PyPreConfig_InitIsolatedConfig()` and `PyConfig_InitIsolatedConfig()` functions create a configuration to isolate Python from the system. For example, to embed Python into an application.

This configuration ignores global configuration variables, environment variables, command line arguments (`PyConfig.argv` is not parsed) and user site directory. The C standard streams (ex: `stdout`) and the LC_CTYPE locale are left unchanged. Signal handlers are not installed.

Configuration files are still used with this configuration to determine paths that are unspecified. Ensure `PyConfig.home` is specified to avoid computing the default path configuration.

10.9 Python Configuration

`PyPreConfig_InitPythonConfig()` and `PyConfig_InitPythonConfig()` functions create a configuration to build a customized Python which behaves as the regular Python.

Environments variables and command line arguments are used to configure Python, whereas global configuration variables are ignored.

This function enables C locale coercion (**PEP 538**) and Python UTF-8 Mode (**PEP 540**) depending on the LC_CTYPE locale, `PYTHONUTF8` and `PYTHONCOERCECLOCALE` environment variables.

10.10 Python Path Configuration

`PyConfig` contains multiple fields for the path configuration:

- Path configuration inputs:
  - `PyConfig.home`
  - `PyConfig.platlibdir`
  - `PyConfig.pathconfig_warnings`
  - `PyConfig.program_name`
  - `PyConfig.pythonpath_env`
  - current working directory: to get absolute paths
  - `PATH` environment variable to get the program full path (from `PyConfig.program_name`)
  - `__PYVENV_LAUNCHER__` environment variable
  - (Windows only) Application paths in the registry under „SoftwarePythonPythonCoreX.YPythonPath” of HKEY_CURRENT_USER and HKEY_LOCAL_MACHINE (where X.Y is the Python version).

- Path configuration output fields:
  - *PyConfig.base_exec_prefix*
  - *PyConfig.base_executable*
  - *PyConfig.base_prefix*
  - *PyConfig.exec_prefix*
  - *PyConfig.executable*
  - *PyConfig.module_search_paths_set*, *PyConfig.module_search_paths*
  - *PyConfig.prefix*

If at least one „output field" is not set, Python calculates the path configuration to fill unset fields. If *module_search_paths_set* is equal to `0`, *module_search_paths* is overridden and *module_search_paths_set* is set to `1`.

It is possible to completely ignore the function calculating the default path configuration by setting explicitly all path configuration output fields listed above. A string is considered as set even if it is non-empty. `module_search_paths` is considered as set if `module_search_paths_set` is set to `1`. In this case, `module_search_paths` will be used without modification.

Set *pathconfig_warnings* to `0` to suppress warnings when calculating the path configuration (Unix only, Windows does not log any warning).

If *base_prefix* or *base_exec_prefix* fields are not set, they inherit their value from *prefix* and *exec_prefix* respectively.

Py_RunMain() and *Py_Main()* modify `sys.path`:

- If *run_filename* is set and is a directory which contains a `__main__.py` script, prepend *run_filename* to `sys.path`.
- If *isolated* is zero:
  - If *run_module* is set, prepend the current directory to `sys.path`. Do nothing if the current directory cannot be read.
  - If *run_filename* is set, prepend the directory of the filename to `sys.path`.
  - Otherwise, prepend an empty string to `sys.path`.

If *site_import* is non-zero, `sys.path` can be modified by the `site` module. If *user_site_directory* is non-zero and the user's site-package directory exists, the `site` module appends the user's site-package directory to `sys.path`.

The following configuration files are used by the path configuration:

- `pyvenv.cfg`
- `._pth` file (ex: `python._pth`)
- `pybuilddir.txt` (Unix only)

If a `._pth` file is present:

- Set *isolated* to `1`.
- Set *use_environment* to `0`.
- Set *site_import* to `0`.
- Set *safe_path* to `1`.

The `__PYVENV_LAUNCHER__` environment variable is used to set *PyConfig.base_executable*.

10.11 Py_GetArgcArgv()

void **Py_GetArgcArgv**(int *argc, wchar_t ***argv)

Get the original command line arguments, before Python modified them.

See also *PyConfig.orig_argv* member.

10.12 Multi-Phase Initialization Private Provisional API

This section is a private provisional API introducing multi-phase initialization, the core feature of **PEP 432**:

- „Core" initialization phase, „bare minimum Python":
  - Builtin types;
  - Builtin exceptions;
  - Builtin and frozen modules;
  - The `sys` module is only partially initialized (ex: `sys.path` doesn't exist yet).
- „Main" initialization phase, Python is fully initialized:
  - Install and configure `importlib`;
  - Apply the *Path Configuration*;
  - Install signal handlers;
  - Finish `sys` module initialization (ex: create `sys.stdout` and `sys.path`);
  - Enable optional features like `faulthandler` and `tracemalloc`;
  - Import the `site` module;
  - etc.

Private provisional API:

- `PyConfig._init_main`: if set to `0`, *Py_InitializeFromConfig()* stops at the „Core" initialization phase.

PyStatus **_Py_InitializeMain**(void)

Move to the „Main" initialization phase, finish the Python initialization.

No module is imported during the „Core" phase and the `importlib` module is not configured: the *Path Configuration* is only applied during the „Main" phase. It may allow to customize Python in Python to override or tune the *Path Configuration*, maybe install a custom `sys.meta_path` importer or an import hook, etc.

It may become possible to calculate the *Path Configuration* in Python, after the Core phase and before the Main phase, which is one of the **PEP 432** motivation.

The „Core" phase is not properly defined: what should be and what should not be available at this phase is not specified yet. The API is marked as private and provisional: the API can be modified or even be removed anytime until a proper public API is designed.

Example running Python code between „Core" and „Main" initialization phases:

```
void init_python(void)
{
    PyStatus status;

    PyConfig config;
    PyConfig_InitPythonConfig(&config);
    config._init_main = 0;
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
    /* ... customize 'config' configuration ... */

    status = Py_InitializeFromConfig(&config);
    PyConfig_Clear(&config);
    if (PyStatus_Exception(status)) {
        Py_ExitStatusException(status);
    }

    /* Use sys.stderr because sys.stdout is only created
       by _Py_InitializeMain() */
    int res = PyRun_SimpleString(
        "import sys; "
        "print('Run Python code before _Py_InitializeMain', "
               "file=sys.stderr)");
    if (res < 0) {
        exit(1);
    }

    /* ... put more configuration code here ... */

    status = _Py_InitializeMain();
    if (PyStatus_Exception(status)) {
        Py_ExitStatusException(status);
    }
}
```

ROZDZIAŁ 11

Zarządzanie Pamięcią

11.1 Skorowidz

Zarządzanie pamięcią w Pythonie zakłada prywatną stertę zawierającą wszystkie obiekty i struktury danych Pythona. Zarządzanie tą prywatną stertą jest zapewniane wewnętrznie przez *zarządcę pamięci Pythona*. Zarządca pamięci Pythona ma różne komponenty które radzą sobie z różnymi aspektami dynamicznego przechowywania, jak współdzielenie, segmentacja, alokacja wstępna i kieszeniowanie.

Na najniższym poziomie, przedmiot przydzielający pamięć „na-surowo" zapewnia że będzie dość pamięci na prywatnej stercie dla przechowania wszystkich związanych-z-językiem-pytonowskim danych przez współdziałanie z zarządcą pamięci systemu operacyjnego. Ponad zarządcą surowej pamięci, kilka szczególnych dla danych typów przedmiotów zarządców operuje na tej samej stercie i wypełnia szczególne zasady zarządzania pamięcią dostosowane do szczególnych własności każdego rodzaju przedmiotu. Dla przykładu przedmioty liczb całkowitych są zarządzane inaczej wewnątrz sterty niż ciągi znaków, krotki czy słowniki gdyż liczby całkowite zakładają inne wymagania przechowywania i wady i zalety prędkości/zajętej przestrzeni. Zarządca pamięcią Pythona zatem odprawia pewną ilość nakładów pracy dla szczególnych dla przedmiotów różnych typów zarządców, ale zapewnia że te drugie będą operować wewnątrz ograniczeń prywatnej sterty.

It is important to understand that the management of the Python heap is performed by the interpreter itself and that the user has no control over it, even if they regularly manipulate object pointers to memory blocks inside that heap. The allocation of heap space for Python objects and other internal buffers is performed on demand by the Python memory manager through the Python/C API functions listed in this document.

To avoid memory corruption, extension writers should never try to operate on Python objects with the functions exported by the C library: `malloc()`, `calloc()`, `realloc()` and `free()`. This will result in mixed calls between the C allocator and the Python memory manager with fatal consequences, because they implement different algorithms and operate on different heaps. However, one may safely allocate and release memory blocks with the C library allocator for individual purposes, as shown in the following example:

```
PyObject *res;
char *buf = (char *) malloc(BUFSIZ); /* for I/O */

if (buf == NULL)
    return PyErr_NoMemory();
...Do some I/O operation involving buf...
res = PyBytes_FromString(buf);
free(buf); /* malloc'ed */
return res;
```

In this example, the memory request for the I/O buffer is handled by the C library allocator. The Python memory manager is involved only in the allocation of the bytes object returned as a result.

In most situations, however, it is recommended to allocate memory from the Python heap specifically because the latter is under control of the Python memory manager. For example, this is required when the interpreter is extended with new object types written in C. Another reason for using the Python heap is the desire to *inform* the Python memory manager about the memory needs of the extension module. Even when the requested memory is used exclusively for internal, highly specific purposes, delegating all memory requests to the Python memory manager causes the interpreter to have a more accurate image of its memory footprint as a whole. Consequently, under certain circumstances, the Python memory manager may or may not trigger appropriate actions, like garbage collection, memory compaction or other preventive procedures. Note that by using the C library allocator as shown in the previous example, the allocated memory for the I/O buffer escapes completely the Python memory manager.

Zobacz także

The `PYTHONMALLOC` environment variable can be used to configure the memory allocators used by Python.

The `PYTHONMALLOCSTATS` environment variable can be used to print statistics of the *pymalloc memory allocator* every time a new pymalloc object arena is created, and on shutdown.

11.2 Allocator Domains

All allocating functions belong to one of three different „domains" (see also `PyMemAllocatorDomain`). These domains represent different allocation strategies and are optimized for different purposes. The specific details on how every domain allocates memory or what internal functions each domain calls is considered an implementation detail, but for debugging purposes a simplified table can be found at *here*. The APIs used to allocate and free a block of memory must be from the same domain. For example, `PyMem_Free()` must be used to free memory allocated using `PyMem_Malloc()`.

The three allocation domains are:

- Raw domain: intended for allocating memory for general-purpose memory buffers where the allocation *must* go to the system allocator or where the allocator can operate without the *GIL*. The memory is requested directly from the system. See *Raw Memory Interface*.
- „Mem" domain: intended for allocating memory for Python buffers and general-purpose memory buffers where the allocation must be performed with the *GIL* held. The memory is taken from the Python private heap. See *Memory Interface*.
- Object domain: intended for allocating memory for Python objects. The memory is taken from the Python private heap. See *Object allocators*.

Informacja

The *free-threaded* build requires that only Python objects are allocated using the „object" domain and that all Python objects are allocated using that domain. This differs from the prior Python versions, where this was only a best practice and not a hard requirement.

For example, buffers (non-Python objects) should be allocated using `PyMem_Malloc()`, `PyMem_RawMalloc()`, or `malloc()`, but not `PyObject_Malloc()`.

See Memory Allocation APIs.

11.3 Raw Memory Interface

The following function sets are wrappers to the system allocator. These functions are thread-safe, the *GIL* does not need to be held.

The *default raw memory allocator* uses the following functions: `malloc()`, `calloc()`, `realloc()` and `free()`; call `malloc(1)` (or `calloc(1, 1)`) when requesting zero bytes.

Added in version 3.4.

void ***PyMem_RawMalloc**(size_t n)

Część stabilnego ABI *od wersji 3.13.* Allocates *n* bytes and returns a pointer of type `void*` to the allocated memory, or `NULL` if the request fails.

Requesting zero bytes returns a distinct non-`NULL` pointer if possible, as if `PyMem_RawMalloc(1)` had been called instead. The memory will not have been initialized in any way.

void ***PyMem_RawCalloc**(size_t nelem, size_t elsize)

Część stabilnego ABI *od wersji 3.13.* Allocates *nelem* elements each whose size in bytes is *elsize* and returns a pointer of type `void*` to the allocated memory, or `NULL` if the request fails. The memory is initialized to zeros.

Requesting zero elements or elements of size zero bytes returns a distinct non-`NULL` pointer if possible, as if `PyMem_RawCalloc(1, 1)` had been called instead.

Added in version 3.5.

void ***PyMem_RawRealloc**(void *p, size_t n)

Część stabilnego ABI *od wersji 3.13.* Resizes the memory block pointed to by *p* to *n* bytes. The contents will be unchanged to the minimum of the old and the new sizes.

If *p* is `NULL`, the call is equivalent to `PyMem_RawMalloc(n)`; else if *n* is equal to zero, the memory block is resized but is not freed, and the returned pointer is non-`NULL`.

Unless *p* is `NULL`, it must have been returned by a previous call to `PyMem_RawMalloc()`, `PyMem_RawRealloc()` or `PyMem_RawCalloc()`.

If the request fails, `PyMem_RawRealloc()` returns `NULL` and *p* remains a valid pointer to the previous memory area.

void **PyMem_RawFree**(void *p)

Część stabilnego ABI *od wersji 3.13.* Frees the memory block pointed to by *p*, which must have been returned by a previous call to `PyMem_RawMalloc()`, `PyMem_RawRealloc()` or `PyMem_RawCalloc()`. Otherwise, or if `PyMem_RawFree(p)` has been called before, undefined behavior occurs.

If *p* is `NULL`, no operation is performed.

11.4 Sprzęg Pamięci

The following function sets, modeled after the ANSI C standard, but specifying behavior when requesting zero bytes, are available for allocating and releasing memory from the Python heap.

The *default memory allocator* uses the *pymalloc memory allocator*.

> **Ostrzeżenie**
>
> The *GIL* must be held when using these functions.

Zmienione w wersji 3.6: The default allocator is now pymalloc instead of system `malloc()`.

void ***PyMem_Malloc**(size_t n)

Część stabilnego ABI. Allocates *n* bytes and returns a pointer of type `void*` to the allocated memory, or `NULL` if the request fails.

Requesting zero bytes returns a distinct non-`NULL` pointer if possible, as if `PyMem_Malloc(1)` had been called instead. The memory will not have been initialized in any way.

void ***PyMem_Calloc**(size_t nelem, size_t elsize)

Część stabilnego ABI *od wersji 3.7.* Allocates *nelem* elements each whose size in bytes is *elsize* and returns a pointer of type `void*` to the allocated memory, or `NULL` if the request fails. The memory is initialized to zeros.

Requesting zero elements or elements of size zero bytes returns a distinct non-`NULL` pointer if possible, as if `PyMem_Calloc(1, 1)` had been called instead.

Added in version 3.5.

void ***PyMem_Realloc**(void *p, size_t n)

Część stabilnego ABI. Resizes the memory block pointed to by *p* to *n* bytes. The contents will be unchanged to the minimum of the old and the new sizes.

If *p* is `NULL`, the call is equivalent to `PyMem_Malloc(n)`; else if *n* is equal to zero, the memory block is resized but is not freed, and the returned pointer is non-`NULL`.

Unless *p* is `NULL`, it must have been returned by a previous call to *PyMem_Malloc()*, *PyMem_Realloc()* or *PyMem_Calloc()*.

If the request fails, *PyMem_Realloc()* returns `NULL` and *p* remains a valid pointer to the previous memory area.

void **PyMem_Free**(void *p)

Część stabilnego ABI. Frees the memory block pointed to by *p*, which must have been returned by a previous call to *PyMem_Malloc()*, *PyMem_Realloc()* or *PyMem_Calloc()*. Otherwise, or if `PyMem_Free(p)` has been called before, undefined behavior occurs.

If *p* is `NULL`, no operation is performed.

Następujące makropolecenia zorientowane-według-typu dostarczone są dla wygody. Zauważ że *TYP* odnosi się do dowolnego typu C.

PyMem_New(TYPE, n)

Same as *PyMem_Malloc()*, but allocates `(n * sizeof(TYPE))` bytes of memory. Returns a pointer cast to `TYPE*`. The memory will not have been initialized in any way.

PyMem_Resize(p, TYPE, n)

Same as *PyMem_Realloc()*, but the memory block is resized to `(n * sizeof(TYPE))` bytes. Returns a pointer cast to `TYPE*`. On return, *p* will be a pointer to the new memory area, or `NULL` in the event of failure.

This is a C preprocessor macro; *p* is always reassigned. Save the original value of *p* to avoid losing memory when handling errors.

void **PyMem_Del**(void *p)

Same as *PyMem_Free()*.

Dodać należy, że następujący zbiór makropoleceń dostarczony jest aby odwoływać się do programu przydzielającego pamięć w języku pytonowskim bezpośrednio, bez udziału zadań sprzęgu C wymienionych powyżej. Jednakże, zauważ, że ich użycie nie zachowuje wzajemnej zgodności binarnej pomiędzy wersjami Pythona i z tego też powodu ich użycie jest niewskazane w modułach rozszerzających.

- `PyMem_MALLOC(size)`
- `PyMem_NEW(type, size)`
- `PyMem_REALLOC(ptr, size)`
- `PyMem_RESIZE(ptr, type, size)`
- `PyMem_FREE(ptr)`
- `PyMem_DEL(ptr)`

11.5 Object allocators

The following function sets, modeled after the ANSI C standard, but specifying behavior when requesting zero bytes, are available for allocating and releasing memory from the Python heap.

> **Informacja**
>
> There is no guarantee that the memory returned by these allocators can be successfully cast to a Python object when intercepting the allocating functions in this domain by the methods described in the *Customize Memory Allocators* section.

The *default object allocator* uses the *pymalloc memory allocator*.

> **Ostrzeżenie**
>
> The *GIL* must be held when using these functions.

void ***PyObject_Malloc**(size_t n)

Część stabilnego ABI. Allocates *n* bytes and returns a pointer of type `void*` to the allocated memory, or `NULL` if the request fails.

Requesting zero bytes returns a distinct non-`NULL` pointer if possible, as if `PyObject_Malloc(1)` had been called instead. The memory will not have been initialized in any way.

void ***PyObject_Calloc**(size_t nelem, size_t elsize)

Część stabilnego ABI *od wersji 3.7.* Allocates *nelem* elements each whose size in bytes is *elsize* and returns a pointer of type `void*` to the allocated memory, or `NULL` if the request fails. The memory is initialized to zeros.

Requesting zero elements or elements of size zero bytes returns a distinct non-`NULL` pointer if possible, as if `PyObject_Calloc(1, 1)` had been called instead.

Added in version 3.5.

void ***PyObject_Realloc**(void *p, size_t n)

Część stabilnego ABI. Resizes the memory block pointed to by *p* to *n* bytes. The contents will be unchanged to the minimum of the old and the new sizes.

If *p* is `NULL`, the call is equivalent to `PyObject_Malloc(n)`; else if *n* is equal to zero, the memory block is resized but is not freed, and the returned pointer is non-`NULL`.

Unless *p* is `NULL`, it must have been returned by a previous call to *`PyObject_Malloc()`*, *`PyObject_Realloc()`* or *`PyObject_Calloc()`*.

If the request fails, *`PyObject_Realloc()`* returns `NULL` and *p* remains a valid pointer to the previous memory area.

void **PyObject_Free**(void *p)

Część stabilnego ABI. Frees the memory block pointed to by *p*, which must have been returned by a previous call to *`PyObject_Malloc()`*, *`PyObject_Realloc()`* or *`PyObject_Calloc()`*. Otherwise, or if `PyObject_Free(p)` has been called before, undefined behavior occurs.

If *p* is `NULL`, no operation is performed.

11.6 Default Memory Allocators

Default memory allocators:

Configuration	Nazwa	PyMem_RawMalloc	PyMem_Malloc	PyObject_Malloc
Release build	`"pymalloc"`	`malloc`	`pymalloc`	`pymalloc`
Debug build	`"pymalloc_debug`	`malloc` + debug	`pymalloc` + debug	`pymalloc` + debug
Release build, without pymalloc	`"malloc"`	`malloc`	`malloc`	`malloc`
Debug build, without pymalloc	`"malloc_debug"`	`malloc` + debug	`malloc` + debug	`malloc` + debug

Legend:

- Name: value for `PYTHONMALLOC` environment variable.
- `malloc`: system allocators from the standard C library, C functions: `malloc()`, `calloc()`, `realloc()` and `free()`.
- `pymalloc`: *pymalloc memory allocator*.
- `mimalloc`: *mimalloc memory allocator*. The pymalloc allocator will be used if mimalloc support isn't available.
- „+ debug": with *debug hooks on the Python memory allocators*.
- „Debug build": Python build in debug mode.

11.7 Customize Memory Allocators

Added in version 3.4.

type **`PyMemAllocatorEx`**

Structure used to describe a memory block allocator. The structure has the following fields:

Pole	Znaczenie
`void *ctx`	user context passed as first argument
`void* malloc(void *ctx, size_t size)`	allocate a memory block
`void* calloc(void *ctx, size_t nelem, size_t elsize)`	allocate a memory block initialized with zeros
`void* realloc(void *ctx, void *ptr, size_t new_size)`	allocate or resize a memory block
`void free(void *ctx, void *ptr)`	free a memory block

Zmienione w wersji 3.5: The `PyMemAllocator` structure was renamed to `PyMemAllocatorEx` and a new `calloc` field was added.

type **`PyMemAllocatorDomain`**

Enum used to identify an allocator domain. Domains:

`PYMEM_DOMAIN_RAW`

Functions:

- `PyMem_RawMalloc()`
- `PyMem_RawRealloc()`
- `PyMem_RawCalloc()`
- `PyMem_RawFree()`

PYMEM_DOMAIN_MEM

Functions:

- `PyMem_Malloc()`,
- `PyMem_Realloc()`
- `PyMem_Calloc()`
- `PyMem_Free()`

PYMEM_DOMAIN_OBJ

Functions:

- `PyObject_Malloc()`
- `PyObject_Realloc()`
- `PyObject_Calloc()`
- `PyObject_Free()`

void **PyMem_GetAllocator**(*PyMemAllocatorDomain* domain, *PyMemAllocatorEx* *allocator)

Get the memory block allocator of the specified domain.

void **PyMem_SetAllocator**(*PyMemAllocatorDomain* domain, *PyMemAllocatorEx* *allocator)

Set the memory block allocator of the specified domain.

The new allocator must return a distinct non-`NULL` pointer when requesting zero bytes.

For the `PYMEM_DOMAIN_RAW` domain, the allocator must be thread-safe: the *GIL* is not held when the allocator is called.

For the remaining domains, the allocator must also be thread-safe: the allocator may be called in different interpreters that do not share a `GIL`.

If the new allocator is not a hook (does not call the previous allocator), the `PyMem_SetupDebugHooks()` function must be called to reinstall the debug hooks on top on the new allocator.

See also `PyPreConfig.allocator` and *Preinitialize Python with PyPreConfig*.

Ostrzeżenie

`PyMem_SetAllocator()` does have the following contract:

- It can be called after `Py_PreInitialize()` and before `Py_InitializeFromConfig()` to install a custom memory allocator. There are no restrictions over the installed allocator other than the ones imposed by the domain (for instance, the Raw Domain allows the allocator to be called without the GIL held). See *the section on allocator domains* for more information.
- If called after Python has finish initializing (after `Py_InitializeFromConfig()` has been called) the allocator **must** wrap the existing allocator. Substituting the current allocator for some other arbitrary one is **not supported**.

Zmienione w wersji 3.12: All allocators must be thread-safe.

void **PyMem_SetupDebugHooks**(void)

Setup *debug hooks in the Python memory allocators* to detect memory errors.

11.8 Debug hooks on the Python memory allocators

When Python is built in debug mode, the `PyMem_SetupDebugHooks()` function is called at the *Python preinitialization* to setup debug hooks on Python memory allocators to detect memory errors.

The `PYTHONMALLOC` environment variable can be used to install debug hooks on a Python compiled in release mode (ex: `PYTHONMALLOC=debug`).

The *PyMem_SetupDebugHooks()* function can be used to set debug hooks after calling *PyMem_SetAllocator()*.

These debug hooks fill dynamically allocated memory blocks with special, recognizable bit patterns. Newly allocated memory is filled with the byte `0xCD` (`PYMEM_CLEANBYTE`), freed memory is filled with the byte `0xDD` (`PYMEM_DEADBYTE`). Memory blocks are surrounded by „forbidden bytes" filled with the byte `0xFD` (`PYMEM_FORBIDDENBYTE`). Strings of these bytes are unlikely to be valid addresses, floats, or ASCII strings.

Runtime checks:

- Detect API violations. For example, detect if *PyObject_Free()* is called on a memory block allocated by *PyMem_Malloc()*.
- Detect write before the start of the buffer (buffer underflow).
- Detect write after the end of the buffer (buffer overflow).
- Check that the *GIL* is held when allocator functions of *PYMEM_DOMAIN_OBJ* (ex: *PyObject_Malloc()*) and *PYMEM_DOMAIN_MEM* (ex: *PyMem_Malloc()*) domains are called.

On error, the debug hooks use the `tracemalloc` module to get the traceback where a memory block was allocated. The traceback is only displayed if `tracemalloc` is tracing Python memory allocations and the memory block was traced.

Let *S* = `sizeof(size_t).2*S` bytes are added at each end of each block of *N* bytes requested. The memory layout is like so, where p represents the address returned by a malloc-like or realloc-like function (`p[i:j]` means the slice of bytes from `*(p+i)` inclusive up to `*(p+j)` exclusive; note that the treatment of negative indices differs from a Python slice):

`p[-2*S:-S]`
Number of bytes originally asked for. This is a size_t, big-endian (easier to read in a memory dump).

`p[-S]`
API identifier (ASCII character):

- `'r'` for *PYMEM_DOMAIN_RAW*.
- `'m'` for *PYMEM_DOMAIN_MEM*.
- `'o'` for *PYMEM_DOMAIN_OBJ*.

`p[-S+1:0]`
Copies of PYMEM_FORBIDDENBYTE. Used to catch under- writes and reads.

`p[0:N]`
The requested memory, filled with copies of PYMEM_CLEANBYTE, used to catch reference to uninitialized memory. When a realloc-like function is called requesting a larger memory block, the new excess bytes are also filled with PYMEM_CLEANBYTE. When a free-like function is called, these are overwritten with PYMEM_DEADBYTE, to catch reference to freed memory. When a realloc- like function is called requesting a smaller memory block, the excess old bytes are also filled with PYMEM_DEADBYTE.

`p[N:N+S]`
Copies of PYMEM_FORBIDDENBYTE. Used to catch over- writes and reads.

`p[N+S:N+2*S]`
Only used if the `PYMEM_DEBUG_SERIALNO` macro is defined (not defined by default).

A serial number, incremented by 1 on each call to a malloc-like or realloc-like function. Big-endian `size_t`. If „bad memory" is detected later, the serial number gives an excellent way to set a breakpoint on the next run, to capture the instant at which this block was passed out. The static function bumpserialno() in obmalloc.c is the only place the serial number is incremented, and exists so you can set such a breakpoint easily.

A realloc-like or free-like function first checks that the PYMEM_FORBIDDENBYTE bytes at each end are intact. If they've been altered, diagnostic output is written to stderr, and the program is aborted via Py_FatalError(). The other main failure mode is provoking a memory error when a program reads up one of the special bit patterns and

tries to use it as an address. If you get in a debugger then and look at the object, you're likely to see that it's entirely filled with PYMEM_DEADBYTE (meaning freed memory is getting used) or PYMEM_CLEANBYTE (meaning uninitialized memory is getting used).

Zmienione w wersji 3.6: The `PyMem_SetupDebugHooks()` function now also works on Python compiled in release mode. On error, the debug hooks now use `tracemalloc` to get the traceback where a memory block was allocated. The debug hooks now also check if the GIL is held when functions of `PYMEM_DOMAIN_OBJ` and `PYMEM_DOMAIN_MEM` domains are called.

Zmienione w wersji 3.8: Byte patterns `0xCB` (`PYMEM_CLEANBYTE`), `0xDB` (`PYMEM_DEADBYTE`) and `0xFB` (`PYMEM_FORBIDDENBYTE`) have been replaced with `0xCD`, `0xDD` and `0xFD` to use the same values than Windows CRT debug `malloc()` and `free()`.

11.9 The pymalloc allocator

Python has a *pymalloc* allocator optimized for small objects (smaller or equal to 512 bytes) with a short lifetime. It uses memory mappings called „arenas" with a fixed size of either 256 KiB on 32-bit platforms or 1 MiB on 64-bit platforms. It falls back to `PyMem_RawMalloc()` and `PyMem_RawRealloc()` for allocations larger than 512 bytes.

pymalloc is the *default allocator* of the `PYMEM_DOMAIN_MEM` (ex: `PyMem_Malloc()`) and `PYMEM_DOMAIN_OBJ` (ex: `PyObject_Malloc()`) domains.

The arena allocator uses the following functions:

- `VirtualAlloc()` and `VirtualFree()` on Windows,
- `mmap()` and `munmap()` if available,
- `malloc()` and `free()` otherwise.

This allocator is disabled if Python is configured with the `--without-pymalloc` option. It can also be disabled at runtime using the `PYTHONMALLOC` environment variable (ex: `PYTHONMALLOC=malloc`).

11.9.1 Customize pymalloc Arena Allocator

Added in version 3.4.

type **`PyObjectArenaAllocator`**

Structure used to describe an arena allocator. The structure has three fields:

Pole	Znaczenie
`void *ctx`	user context passed as first argument
`void* alloc(void *ctx, size_t size)`	allocate an arena of size bytes
`void free(void *ctx, void *ptr, size_t size)`	free an arena

void **`PyObject_GetArenaAllocator`**(*PyObjectArenaAllocator* *allocator)

Get the arena allocator.

void **`PyObject_SetArenaAllocator`**(*PyObjectArenaAllocator* *allocator)

Set the arena allocator.

11.10 The mimalloc allocator

Added in version 3.13.

Python supports the mimalloc allocator when the underlying platform support is available. mimalloc „is a general purpose allocator with excellent performance characteristics. Initially developed by Daan Leijen for the runtime systems of the Koka and Lean languages."

11.11 tracemalloc C API

Added in version 3.7.

int **PyTraceMalloc_Track**(unsigned int domain, uintptr_t ptr, size_t size)

Track an allocated memory block in the `tracemalloc` module.

Return `0` on success, return `-1` on error (failed to allocate memory to store the trace). Return `-2` if tracemalloc is disabled.

If memory block is already tracked, update the existing trace.

int **PyTraceMalloc_Untrack**(unsigned int domain, uintptr_t ptr)

Untrack an allocated memory block in the `tracemalloc` module. Do nothing if the block was not tracked.

Return `-2` if tracemalloc is disabled, otherwise return `0`.

11.12 Przykłady

Tutaj jest przykład z sekcji „przeglądu pamięci" - z ang. - *Skorowidz*, przepisane, tak aby przestrzeń wejścia/wyjścia była przydzielona ze sterty Pythona używając pierwszego zestawu zadań:

```
PyObject *res;
char *buf = (char *) PyMem_Malloc(BUFSIZ); /* for I/O */

if (buf == NULL)
    return PyErr_NoMemory();
/* ...Do some I/O operation involving buf... */
res = PyBytes_FromString(buf);
PyMem_Free(buf); /* allocated with PyMem_Malloc */
return res;
```

ten sam kod przy użyciu zorientowanych na typ zbiorów zadań:

```
PyObject *res;
char *buf = PyMem_New(char, BUFSIZ); /* for I/O */

if (buf == NULL)
    return PyErr_NoMemory();
/* ...Do some I/O operation involving buf... */
res = PyBytes_FromString(buf);
PyMem_Del(buf); /* allocated with PyMem_New */
return res;
```

Zauważ, że w dwóch powyższych przykładach, przestrzeń wymiany jest zawsze zmieniana przez zadania należące do tego samego zbioru. Właściwie, jest wymagane użycie tej samej rodziny sprzęgów zarządzania pamięcią (z ang. - memory API) dla danego obszaru pamięci, tak, że ryzyko pomieszania różnych programów lokujących zmniejszone jest do minimum. Następująca sekwencja zawiera dwa błędy, jeden z których określony jest jako *krytyczny* ponieważ miesza dwa różne programy lokujące pamięć działające na różnych stertach.

```
char *buf1 = PyMem_New(char, BUFSIZ);
char *buf2 = (char *) malloc(BUFSIZ);
char *buf3 = (char *) PyMem_Malloc(BUFSIZ);
...
PyMem_Del(buf3);  /* Wrong -- should be PyMem_Free() */
free(buf2);       /* Right -- allocated via malloc() */
free(buf1);       /* Fatal -- should be PyMem_Del()  */
```

In addition to the functions aimed at handling raw memory blocks from the Python heap, objects in Python are allocated and released with `PyObject_New`, `PyObject_NewVar` and `PyObject_Del()`.

Te zostaną wyjaśnione w następnym rozdziale o określaniu i realizowaniu nowych typów obiektów w języku C.

ROZDZIAŁ 12

Object Implementation Support

This chapter describes the functions, types, and macros used when defining new object types.

12.1 Przydzielanie obiektów na stercie

PyObject ***_PyObject_New**(*PyTypeObject* *type)

Wartość zwracana: nowa referencja.

PyVarObject ***_PyObject_NewVar**(*PyTypeObject* *type, *Py_ssize_t* size)

Wartość zwracana: nowa referencja.

PyObject ***PyObject_Init**(*PyObject* *op, *PyTypeObject* *type)

Wartość zwracana: pożyczona referencja. Część stabilnego ABI. Inicjalizacja nowo przydzielonego obiektu o nazwie *op*, określa jego typ oraz początkowe odniesienie. Funkcja zwraca zainicjowany obiekt. Jeżeli wartość *type* wskazuje, że obiekt bierze udział w detekcji cyklicznych zbędnych danych „garbage detector", to zostaje on dodany do zbioru obiektów obserwowanych przez detektor. Pozostałe pola obiektu pozostają niezmienione.

PyVarObject ***PyObject_InitVar**(*PyVarObject* *op, *PyTypeObject* *type, *Py_ssize_t* size)

Wartość zwracana: pożyczona referencja. Część stabilnego ABI. To wszystko, czym zajmie się funkcja `PyObject_Init()`, a ponadto zainicjuje informacje o długości dla obiektu o zmiennym rozmiarze.

PyObject_New(TYPE, typeobj)

Allocate a new Python object using the C structure type *TYPE* and the Python type object *typeobj* (`PyTypeObject*`). Fields not defined by the Python object header are not initialized. The caller will own the only reference to the object (i.e. its reference count will be one). The size of the memory allocation is determined from the `tp_basicsize` field of the type object.

PyObject_NewVar(TYPE, typeobj, size)

Allocate a new Python object using the C structure type *TYPE* and the Python type object *typeobj* (`PyTypeObject*`). Fields not defined by the Python object header are not initialized. The allocated memory allows for the *TYPE* structure plus *size* (`Py_ssize_t`) fields of the size given by the `tp_itemsize` field of *typeobj*. This is useful for implementing objects like tuples, which are able to determine their size at construction time. Embedding the array of fields into the same allocation decreases the number of allocations, improving the memory management efficiency.

void **PyObject_Del**(void *op)

Releases memory allocated to an object using `PyObject_New` or `PyObject_NewVar`. This is normally called

from the `tp_dealloc` handler specified in the object's type. The fields of the object should not be accessed after this call as the memory is no longer a valid Python object.

PyObject **_Py_NoneStruct**

Obiekt widoczny w Pythonie jako `None`. Dostęp do tego należy uzyskać wyłącznie za pomocą makra `Py_None`, którego wartością jest wskaźnik do tego obiektu.

Zobacz także

PyModule_Create()
Przydzielanie i tworzenie modułów rozszerzeń.

12.2 Wspólne struktury obiektów

Istnieje duża liczba struktur, które są używane przy definiowaniu rodzajów obiektów w Pythonie. Ten rozdział opisuje te struktury i jak są one używane.

12.2.1 Base object types and macros

All Python objects ultimately share a small number of fields at the beginning of the object's representation in memory. These are represented by the `PyObject` and `PyVarObject` types, which are defined, in turn, by the expansions of some macros also used, whether directly or indirectly, in the definition of all other Python objects. Additional macros can be found under *reference counting*.

type **PyObject**

Część ograniczonego API. *(tylko niektóre składniki są częścią stabilnego ABI)* All object types are extensions of this type. This is a type which contains the information Python needs to treat a pointer to an object as an object. In a normal „release" build, it contains only the object's reference count and a pointer to the corresponding type object. Nothing is actually declared to be a `PyObject`, but every pointer to a Python object can be cast to a `PyObject*`. Access to the members must be done by using the macros `Py_REFCNT` and `Py_TYPE`.

type **PyVarObject**

Część ograniczonego API. *(tylko niektóre składniki są częścią stabilnego ABI)* This is an extension of `PyObject` that adds the `ob_size` field. This is only used for objects that have some notion of *length*. This type does not often appear in the Python/C API. Access to the members must be done by using the macros `Py_REFCNT`, `Py_TYPE`, and `Py_SIZE`.

PyObject_HEAD

This is a macro used when declaring new types which represent objects without a varying length. The PyObject_HEAD macro expands to:

```
PyObject ob_base;
```

See documentation of `PyObject` above.

PyObject_VAR_HEAD

This is a macro used when declaring new types which represent objects with a length that varies from instance to instance. The PyObject_VAR_HEAD macro expands to:

```
PyVarObject ob_base;
```

See documentation of `PyVarObject` above.

int **Py_Is** (*PyObject* *x, *PyObject* *y)

Część stabilnego ABI *od wersji 3.10.* Test if the *x* object is the *y* object, the same as `x is y` in Python.

Added in version 3.10.

int **Py_IsNone**(*PyObject* *x)

Część stabilnego ABI *od wersji 3.10.* Test if an object is the `None` singleton, the same as `x is None` in Python.

Added in version 3.10.

int **Py_IsTrue**(*PyObject* *x)

Część stabilnego ABI *od wersji 3.10.* Test if an object is the `True` singleton, the same as `x is True` in Python.

Added in version 3.10.

int **Py_IsFalse**(*PyObject* *x)

Część stabilnego ABI *od wersji 3.10.* Test if an object is the `False` singleton, the same as `x is False` in Python.

Added in version 3.10.

PyTypeObject ***Py_TYPE**(*PyObject* *o)

Wartość zwracana: pożyczona referencja. Get the type of the Python object *o*.

Return a *borrowed reference*.

Use the `Py_SET_TYPE()` function to set an object type.

Zmienione w wersji 3.11: `Py_TYPE()` is changed to an inline static function. The parameter type is no longer `const PyObject*`.

int **Py_IS_TYPE**(*PyObject* *o, *PyTypeObject* *type)

Return non-zero if the object *o* type is *type*. Return zero otherwise. Equivalent to: `Py_TYPE(o) == type`.

Added in version 3.9.

void **Py_SET_TYPE**(*PyObject* *o, *PyTypeObject* *type)

Set the object *o* type to *type*.

Added in version 3.9.

Py_ssize_t **Py_SIZE**(*PyVarObject* *o)

Get the size of the Python object *o*.

Use the `Py_SET_SIZE()` function to set an object size.

Zmienione w wersji 3.11: `Py_SIZE()` is changed to an inline static function. The parameter type is no longer `const PyVarObject*`.

void **Py_SET_SIZE**(*PyVarObject* *o, *Py_ssize_t* size)

Set the object *o* size to *size*.

Added in version 3.9.

PyObject_HEAD_INIT(type)

This is a macro which expands to initialization values for a new `PyObject` type. This macro expands to:

```
_PyObject_EXTRA_INIT
1, type,
```

PyVarObject_HEAD_INIT(type, size)

This is a macro which expands to initialization values for a new `PyVarObject` type, including the `ob_size` field. This macro expands to:

```
_PyObject_EXTRA_INIT
1, type, size,
```

12.2.2 Implementing functions and methods

type **PyCFunction**

Część stabilnego ABI. Type of the functions used to implement most Python callables in C. Functions of this type take two *PyObject** parameters and return one such value. If the return value is NULL, an exception shall have been set. If not NULL, the return value is interpreted as the return value of the function as exposed in Python. The function must return a new reference.

The function signature is:

```
PyObject *PyCFunction(PyObject *self,
                      PyObject *args);
```

type **PyCFunctionWithKeywords**

Część stabilnego ABI. Type of the functions used to implement Python callables in C with signature *METH_VARARGS | METH_KEYWORDS*. The function signature is:

```
PyObject *PyCFunctionWithKeywords(PyObject *self,
                                  PyObject *args,
                                  PyObject *kwargs);
```

type **PyCFunctionFast**

Część stabilnego ABI *od wersji 3.13.* Type of the functions used to implement Python callables in C with signature *METH_FASTCALL*. The function signature is:

```
PyObject *PyCFunctionFast(PyObject *self,
                          PyObject *const *args,
                          Py_ssize_t nargs);
```

type **PyCFunctionFastWithKeywords**

Część stabilnego ABI *od wersji 3.13.* Type of the functions used to implement Python callables in C with signature *METH_FASTCALL | METH_KEYWORDS*. The function signature is:

```
PyObject *PyCFunctionFastWithKeywords(PyObject *self,
                                      PyObject *const *args,
                                      Py_ssize_t nargs,
                                      PyObject *kwnames);
```

type **PyCMethod**

Type of the functions used to implement Python callables in C with signature *METH_METHOD | METH_FASTCALL | METH_KEYWORDS*. The function signature is:

```
PyObject *PyCMethod(PyObject *self,
                    PyTypeObject *defining_class,
                    PyObject *const *args,
                    Py_ssize_t nargs,
                    PyObject *kwnames)
```

Added in version 3.9.

type **PyMethodDef**

Część stabilnego ABI *(w tym wszystkie składniki).* Structure used to describe a method of an extension type. This structure has four fields:

const char ***ml_name**

Name of the method.

PyCFunction **ml_meth**

Pointer to the C implementation.

int **ml_flags**

Flags bits indicating how the call should be constructed.

const char ***ml_doc**

Points to the contents of the docstring.

The `ml_meth` is a C function pointer. The functions may be of different types, but they always return `PyObject*`. If the function is not of the `PyCFunction`, the compiler will require a cast in the method table. Even though `PyCFunction` defines the first parameter as `PyObject*`, it is common that the method implementation uses the specific C type of the *self* object.

The `ml_flags` field is a bitfield which can include the following flags. The individual flags indicate either a calling convention or a binding convention.

There are these calling conventions:

METH_VARARGS

This is the typical calling convention, where the methods have the type `PyCFunction`. The function expects two `PyObject*` values. The first one is the *self* object for methods; for module functions, it is the module object. The second parameter (often called *args*) is a tuple object representing all arguments. This parameter is typically processed using `PyArg_ParseTuple()` or `PyArg_UnpackTuple()`.

METH_KEYWORDS

Can only be used in certain combinations with other flags: *METH_VARARGS | METH_KEYWORDS*, *METH_FASTCALL | METH_KEYWORDS* and *METH_METHOD | METH_FASTCALL | METH_KEYWORDS*.

METH_VARARGS | METH_KEYWORDS

Methods with these flags must be of type `PyCFunctionWithKeywords`. The function expects three parameters: *self*, *args*, *kwargs* where *kwargs* is a dictionary of all the keyword arguments or possibly `NULL` if there are no keyword arguments. The parameters are typically processed using `PyArg_ParseTupleAndKeywords()`.

METH_FASTCALL

Fast calling convention supporting only positional arguments. The methods have the type `PyCFunctionFast`. The first parameter is *self*, the second parameter is a C array of `PyObject*` values indicating the arguments and the third parameter is the number of arguments (the length of the array).

Added in version 3.7.

Zmienione w wersji 3.10: `METH_FASTCALL` is now part of the *stable ABI*.

METH_FASTCALL | METH_KEYWORDS

Extension of `METH_FASTCALL` supporting also keyword arguments, with methods of type `PyCFunctionFastWithKeywords`. Keyword arguments are passed the same way as in the *vectorcall protocol*: there is an additional fourth `PyObject*` parameter which is a tuple representing the names of the keyword arguments (which are guaranteed to be strings) or possibly `NULL` if there are no keywords. The values of the keyword arguments are stored in the *args* array, after the positional arguments.

Added in version 3.7.

METH_METHOD

Can only be used in the combination with other flags: *METH_METHOD | METH_FASTCALL | METH_KEYWORDS*.

METH_METHOD | METH_FASTCALL | METH_KEYWORDS

Extension of *METH_FASTCALL | METH_KEYWORDS* supporting the *defining class*, that is, the class that contains the method in question. The defining class might be a superclass of `Py_TYPE(self)`.

The method needs to be of type `PyCMethod`, the same as for `METH_FASTCALL | METH_KEYWORDS` with `defining_class` argument added after `self`.

Added in version 3.9.

METH_NOARGS

Methods without parameters don't need to check whether arguments are given if they are listed with the

METH_NOARGS flag. They need to be of type *PyCFunction*. The first parameter is typically named *self* and will hold a reference to the module or object instance. In all cases the second parameter will be NULL.

The function must have 2 parameters. Since the second parameter is unused, *Py_UNUSED* can be used to prevent a compiler warning.

METH_O

Methods with a single object argument can be listed with the *METH_O* flag, instead of invoking *PyArg_ParseTuple()* with a "O" argument. They have the type *PyCFunction*, with the *self* parameter, and a *PyObject** parameter representing the single argument.

Te dwie stałe nie są używane do zaznaczania konwencji wywoływania, ale wiążą gdy są używane z metodami klas. Nie mogą być one używane dla funkcji określonych dla modułów. Co najwyżej jedna z tych flag może być ustawiona dla dowolnej danej metody.

METH_CLASS

Metodzie zostanie przekazany typ obiektu jako pierwszy parametr zamiast instancji tego typu. Jest to używane aby tworzyć *metody klasowe*, podobnie do tego, co jest tworzone przy użyciu wbudowanej funkcji classmethod().

METH_STATIC

The method will be passed NULL as the first parameter rather than an instance of the type. This is used to create *static methods*, similar to what is created when using the staticmethod() built-in function.

Jedna pozostała stała kontroluje czy metoda jest ładowana w miejscu innej definicji o tej samej nazwie metody.

METH_COEXIST

The method will be loaded in place of existing definitions. Without *METH_COEXIST*, the default is to skip repeated definitions. Since slot wrappers are loaded before the method table, the existence of a *sq_contains* slot, for example, would generate a wrapped method named __contains__() and preclude the loading of a corresponding PyCFunction with the same name. With the flag defined, the PyCFunction will be loaded in place of the wrapper object and will co-exist with the slot. This is helpful because calls to PyCFunctions are optimized more than wrapper object calls.

PyObject ***PyCMethod_New**(*PyMethodDef* *ml, *PyObject* *self, *PyObject* *module, *PyTypeObject* *cls)

Wartość zwracana: nowa referencja. Część stabilnego ABI *od wersji 3.9.* Turn *ml* into a Python *callable* object. The caller must ensure that *ml* outlives the *callable*. Typically, *ml* is defined as a static variable.

The *self* parameter will be passed as the *self* argument to the C function in ml->ml_meth when invoked. *self* can be NULL.

The *callable* object's __module__ attribute can be set from the given *module* argument. *module* should be a Python string, which will be used as name of the module the function is defined in. If unavailable, it can be set to None or NULL.

Zobacz także

function.__module__

The *cls* parameter will be passed as the *defining_class* argument to the C function. Must be set if *METH_METHOD* is set on ml->ml_flags.

Added in version 3.9.

PyObject ***PyCFunction_NewEx**(*PyMethodDef* *ml, *PyObject* *self, *PyObject* *module)

Wartość zwracana: nowa referencja. Część stabilnego ABI. Equivalent to PyCMethod_New(ml, self, module, NULL).

PyObject ***PyCFunction_New**(*PyMethodDef* *ml, *PyObject* *self)

Wartość zwracana: nowa referencja. Część stabilnego ABI *od wersji 3.4.* Equivalent to PyCMethod_New(ml, self, NULL, NULL).

12.2.3 Accessing attributes of extension types

type **PyMemberDef**

Część stabilnego ABI *(w tym wszystkie składniki).* Structure which describes an attribute of a type which corresponds to a C struct member. When defining a class, put a NULL-terminated array of these structures in the `tp_members` slot.

Its fields are, in order:

const char ***name**

Name of the member. A NULL value marks the end of a `PyMemberDef[]` array.

The string should be static, no copy is made of it.

int **type**

The type of the member in the C struct. See *Member types* for the possible values.

Py_ssize_t **offset**

The offset in bytes that the member is located on the type's object struct.

int **flags**

Zero or more of the *Member flags*, combined using bitwise OR.

const char ***doc**

The docstring, or NULL. The string should be static, no copy is made of it. Typically, it is defined using `PyDoc_STR`.

By default (when `flags` is `0`), members allow both read and write access. Use the `Py_READONLY` flag for read-only access. Certain types, like `Py_T_STRING`, imply `Py_READONLY`. Only `Py_T_OBJECT_EX` (and legacy `T_OBJECT`) members can be deleted.

For heap-allocated types (created using `PyType_FromSpec()` or similar), `PyMemberDef` may contain a definition for the special member `"__vectorcalloffset__"`, corresponding to `tp_vectorcall_offset` in type objects. These must be defined with `Py_T_PYSSIZET` and `Py_READONLY`, for example:

```
static PyMemberDef spam_type_members[] = {
    {"__vectorcalloffset__", Py_T_PYSSIZET,
     offsetof(Spam_object, vectorcall), Py_READONLY},
    {NULL}  /* Sentinel */
};
```

(You may need to `#include <stddef.h>` for `offsetof()`.)

The legacy offsets `tp_dictoffset` and `tp_weaklistoffset` can be defined similarly using `"__dictoffset__"` and `"__weaklistoffset__"` members, but extensions are strongly encouraged to use `Py_TPFLAGS_MANAGED_DICT` and `Py_TPFLAGS_MANAGED_WEAKREF` instead.

Zmienione w wersji 3.12: `PyMemberDef` is always available. Previously, it required including `"structmember.h"`.

PyObject ***PyMember_GetOne**(const char *obj_addr, struct *PyMemberDef* *m)

Część stabilnego ABI. Get an attribute belonging to the object at address *obj_addr*. The attribute is described by `PyMemberDef` *m*. Returns `NULL` on error.

Zmienione w wersji 3.12: `PyMember_GetOne` is always available. Previously, it required including `"structmember.h"`.

int **PyMember_SetOne**(char *obj_addr, struct *PyMemberDef* *m, *PyObject* *o)

Część stabilnego ABI. Set an attribute belonging to the object at address *obj_addr* to object *o*. The attribute to set is described by `PyMemberDef` *m*. Returns `0` if successful and a negative value on failure.

Zmienione w wersji 3.12: `PyMember_SetOne` is always available. Previously, it required including `"structmember.h"`.

Member flags

The following flags can be used with *PyMemberDef.flags*:

Py_READONLY

Not writable.

Py_AUDIT_READ

Emit an `object.__getattr__` audit event before reading.

Py_RELATIVE_OFFSET

Indicates that the *offset* of this `PyMemberDef` entry indicates an offset from the subclass-specific data, rather than from `PyObject`.

Can only be used as part of *Py_tp_members slot* when creating a class using negative *basicsize*. It is mandatory in that case.

This flag is only used in *PyType_Slot*. When setting *tp_members* during class creation, Python clears it and sets *PyMemberDef.offset* to the offset from the `PyObject` struct.

Zmienione w wersji 3.10: The `RESTRICTED`, `READ_RESTRICTED` and `WRITE_RESTRICTED` macros available with `#include "structmember.h"` are deprecated. `READ_RESTRICTED` and `RESTRICTED` are equivalent to *Py_AUDIT_READ*; `WRITE_RESTRICTED` does nothing.

Zmienione w wersji 3.12: The `READONLY` macro was renamed to *Py_READONLY*. The `PY_AUDIT_READ` macro was renamed with the `Py_` prefix. The new names are now always available. Previously, these required `#include "structmember.h"`. The header is still available and it provides the old names.

Member types

PyMemberDef.type can be one of the following macros corresponding to various C types. When the member is accessed in Python, it will be converted to the equivalent Python type. When it is set from Python, it will be converted back to the C type. If that is not possible, an exception such as `TypeError` or `ValueError` is raised.

Unless marked (D), attributes defined this way cannot be deleted using e.g. `del` or `delattr()`.

Macro name	C type	Python type
Py_T_BYTE	`char`	`int`
Py_T_SHORT	`short`	`int`
Py_T_INT	`int`	`int`
Py_T_LONG	`long`	`int`
Py_T_LONGLONG	`long long`	`int`
Py_T_UBYTE	`unsigned char`	`int`
Py_T_UINT	`unsigned int`	`int`
Py_T_USHORT	`unsigned short`	`int`
Py_T_ULONG	`unsigned long`	`int`
Py_T_ULONGLONG	`unsigned long long`	`int`
Py_T_PYSSIZET	*Py_ssize_t*	`int`
Py_T_FLOAT	`float`	`float`
Py_T_DOUBLE	`double`	`float`
Py_T_BOOL	`char` (written as 0 or 1)	`bool`
Py_T_STRING	`const char*` (*)	`str` (RO)
Py_T_STRING_INPLACE	`const char[]` (*)	`str` (RO)
Py_T_CHAR	`char` (0-127)	`str` (**)
Py_T_OBJECT_EX	*PyObject**	`object` (D)

(*): Zero-terminated, UTF8-encoded C string. With `Py_T_STRING` the C representation is a pointer; with `Py_T_STRING_INPLACE` the string is stored directly in the structure.

(**): String of length 1. Only ASCII is accepted.

(RO): Implies *Py_READONLY*.

(D): Can be deleted, in which case the pointer is set to NULL. Reading a NULL pointer raises AttributeError.

Added in version 3.12: In previous versions, the macros were only available with `#include "structmember.h"` and were named without the Py_ prefix (e.g. as T_INT). The header is still available and contains the old names, along with the following deprecated types:

T_OBJECT

Like Py_T_OBJECT_EX, but NULL is converted to None. This results in surprising behavior in Python: deleting the attribute effectively sets it to None.

T_NONE

Always None. Must be used with *Py_READONLY*.

Defining Getters and Setters

type **PyGetSetDef**

Część stabilnego ABI *(w tym wszystkie składniki).* Structure to define property-like access for a type. See also description of the *PyTypeObject.tp_getset* slot.

const char ***name**

attribute name

getter **get**

C function to get the attribute.

setter **set**

Optional C function to set or delete the attribute. If NULL, the attribute is read-only.

const char ***doc**

optional docstring

void ***closure**

Optional user data pointer, providing additional data for getter and setter.

typedef *PyObject* *(***getter**)(*PyObject**, void*)

Część stabilnego ABI. The get function takes one *PyObject** parameter (the instance) and a user data pointer (the associated closure):

It should return a new reference on success or NULL with a set exception on failure.

typedef int (***setter**)(*PyObject**, *PyObject**, void*)

Część stabilnego ABI. set functions take two *PyObject** parameters (the instance and the value to be set) and a user data pointer (the associated closure):

In case the attribute should be deleted the second parameter is NULL. Should return 0 on success or -1 with a set exception on failure.

12.3 Type Objects

Perhaps one of the most important structures of the Python object system is the structure that defines a new type: the *PyTypeObject* structure. Type objects can be handled using any of the PyObject_* or PyType_* functions, but do not offer much that's interesting to most Python applications. These objects are fundamental to how objects behave, so they are very important to the interpreter itself and to any extension module that implements new types.

Type objects are fairly large compared to most of the standard types. The reason for the size is that each type object stores a large number of values, mostly C function pointers, each of which implements a small part of the type's functionality. The fields of the type object are examined in detail in this section. The fields will be described in the order in which they occur in the structure.

In addition to the following quick reference, the *Przykłady* section provides at-a-glance insight into the meaning and use of `PyTypeObject`.

12.3.1 Quick Reference

„tp slots”

PyTypeObject Slot[Strona 274, 1]	*Type*	special methods/attrs	Info[Strona 274, 2]			
			C	T	D	I
<R> `tp_name`	const char *	__name__	X	X		
`tp_basicsize`	`Py_ssize_t`		X	X		X
`tp_itemsize`	`Py_ssize_t`			X		X
`tp_dealloc`	`destructor`		X	X		X
`tp_vectorcall_offset`	`Py_ssize_t`			X		X
(`tp_getattr`)	`getattrfunc`	__getattribute__, __getattr__				G
(`tp_setattr`)	`setattrfunc`	__setattr__, __delattr__				G
`tp_as_async`	`PyAsyncMethods` *	*sub-slots*				%
`tp_repr`	`reprfunc`	__repr__	X	X		X
`tp_as_number`	`PyNumberMethods` *	*sub-slots*				%
`tp_as_sequence`	`PySequenceMethods` *	*sub-slots*				%
`tp_as_mapping`	`PyMappingMethods` *	*sub-slots*				%
`tp_hash`	`hashfunc`	__hash__	X			G
`tp_call`	`ternaryfunc`	__call__		X		X
`tp_str`	`reprfunc`	__str__	X			X
`tp_getattro`	`getattrofunc`	__getattribute__, __getattr__	X	X		G
`tp_setattro`	`setattrofunc`	__setattr__, __delattr__	X	X		G
`tp_as_buffer`	`PyBufferProcs` *					%
`tp_flags`	nieoznaczony typ długi		X	X		?
`tp_doc`	const char *	__doc__	X	X		
`tp_traverse`	`traverseproc`			X		G
`tp_clear`	`inquiry`			X		G
`tp_richcompare`	`richcmpfunc`	__lt__, __le__, __eq__, __ne__, __gt__, __ge__	X			G
(`tp_weaklistoffset`)	`Py_ssize_t`			X		?
`tp_iter`	`getiterfunc`	__iter__				X
`tp_iternext`	`iternextfunc`	__next__				X
`tp_methods`	`PyMethodDef` []		X	X		
`tp_members`	`PyMemberDef` []			X		
`tp_getset`	`PyGetSetDef` []		X	X		
`tp_base`	`PyTypeObject` *	__base__			X	
`tp_dict`	`PyObject` *	__dict__			?	
`tp_descr_get`	`descrgetfunc`	__get__				X
`tp_descr_set`	`descrsetfunc`	__set__, __delete__				X
(`tp_dictoffset`)	`Py_ssize_t`			X		?
`tp_init`	`initproc`	__init__	X	X		X
`tp_alloc`	`allocfunc`		X		?	?
`tp_new`	`newfunc`	__new__	X	X	?	?
`tp_free`	`freefunc`		X	X	?	?
`tp_is_gc`	`inquiry`			X		X
<`tp_bases`>	`PyObject` *	__bases__			~	
<`tp_mro`>	`PyObject` *	__mro__			~	
[`tp_cache`]	`PyObject` *					
[`tp_subclasses`]	void *	__subclasses__				
[`tp_weaklist`]	`PyObject` *					
(`tp_del`)	`destructor`					

ciąg dalszy na następnej stronie

Tabela 1 – kontynuacja poprzedniej strony

PyTypeObject Slot[1]	*Type*	special methods/attrs	Info[2] C	T	D	I
[*tp_version_tag*]	nieoznaczony typ int					
tp_finalize	*destructor*	__del__				X
tp_vectorcall	*vectorcallfunc*					
[*tp_watched*]	nieoznaczony typ znakowy					

sub-slots

Slot	*Type*	special methods
am_await	*unaryfunc*	__await__
am_aiter	*unaryfunc*	__aiter__
am_anext	*unaryfunc*	__anext__
am_send	*sendfunc*	
nb_add	*binaryfunc*	__add__ __radd__
nb_inplace_add	*binaryfunc*	__iadd__
nb_subtract	*binaryfunc*	__sub__ __rsub__
nb_inplace_subtract	*binaryfunc*	__isub__
nb_multiply	*binaryfunc*	__mul__ __rmul__
nb_inplace_multiply	*binaryfunc*	__imul__
nb_remainder	*binaryfunc*	__mod__ __rmod__
nb_inplace_remainder	*binaryfunc*	__imod__
nb_divmod	*binaryfunc*	__divmod__ __rdivmod__
nb_power	*ternaryfunc*	__pow__ __rpow__
nb_inplace_power	*ternaryfunc*	__ipow__
nb_negative	*unaryfunc*	__neg__
nb_positive	*unaryfunc*	__pos__
nb_absolute	*unaryfunc*	__abs__
nb_bool	*inquiry*	__bool__
nb_invert	*unaryfunc*	__invert__
nb_lshift	*binaryfunc*	__lshift__ __rlshift__
nb_inplace_lshift	*binaryfunc*	__ilshift__

ciąg dalszy na następnej stronie

[1] (): A slot name in parentheses indicates it is (effectively) deprecated.
<>: Names in angle brackets should be initially set to `NULL` and treated as read-only.
[]: Names in square brackets are for internal use only.
<R> (as a prefix) means the field is required (must be non-`NULL`).
[2] Columns:
„O”: set on `PyBaseObject_Type`
„T”: set on *`PyType_Type`*
„D”: default (if slot is set to `NULL`)

```
X - PyType_Ready sets this value if it is NULL
~ - PyType_Ready always sets this value (it should be NULL)
? - PyType_Ready may set this value depending on other slots

Also see the inheritance column ("I").
```

„I”: inheritance

```
X - type slot is inherited via *PyType_Ready* if defined with a *NULL* value
% - the slots of the sub-struct are inherited individually
G - inherited, but only in combination with other slots; see the slot's description
? - it's complicated; see the slot's description
```

Note that some slots are effectively inherited through the normal attribute lookup chain.

Tabela 2 – kontynuacja poprzedniej strony

Slot	*Type*	special methods
nb_rshift	*binaryfunc*	__rshift__ __rrshift__
nb_inplace_rshift	*binaryfunc*	__irshift__
nb_and	*binaryfunc*	__and__ __rand__
nb_inplace_and	*binaryfunc*	__iand__
nb_xor	*binaryfunc*	__xor__ __rxor__
nb_inplace_xor	*binaryfunc*	__ixor__
nb_or	*binaryfunc*	__or__ __ror__
nb_inplace_or	*binaryfunc*	__ior__
nb_int	*unaryfunc*	__int__
nb_reserved	void *	
nb_float	*unaryfunc*	__float__
nb_floor_divide	*binaryfunc*	__floordiv__
nb_inplace_floor_divide	*binaryfunc*	__ifloordiv__
nb_true_divide	*binaryfunc*	__truediv__
nb_inplace_true_divide	*binaryfunc*	__itruediv__
nb_index	*unaryfunc*	__index__
nb_matrix_multiply	*binaryfunc*	__matmul__ __rmatmul__
nb_inplace_matrix_multiply	*binaryfunc*	__imatmul__
mp_length	*lenfunc*	__len__
mp_subscript	*binaryfunc*	__getitem__
mp_ass_subscript	*objobjargproc*	__setitem__, __delitem__
sq_length	*lenfunc*	__len__
sq_concat	*binaryfunc*	__add__
sq_repeat	*ssizeargfunc*	__mul__
sq_item	*ssizeargfunc*	__getitem__
sq_ass_item	*ssizeobjargproc*	__setitem__ __delitem__
sq_contains	*objobjproc*	__contains__
sq_inplace_concat	*binaryfunc*	__iadd__
sq_inplace_repeat	*ssizeargfunc*	__imul__
bf_getbuffer	*getbufferproc()*	
bf_releasebuffer	*releasebufferproc()*	

slot typedefs

typedef	Parameter Types	Return Type
allocfunc	PyTypeObject * Py_ssize_t	PyObject *
destructor	PyObject *	void
freefunc	void *	void
traverseproc	PyObject * visitproc void *	int
newfunc	PyObject * PyObject * PyObject *	PyObject *
initproc	PyObject * PyObject * PyObject *	int
reprfunc	PyObject *	PyObject *
getattrfunc	PyObject * const char *	PyObject *
setattrfunc	PyObject * const char * PyObject *	int
getattrofunc	PyObject * PyObject *	PyObject *
setattrofunc	PyObject * PyObject * PyObject *	int
descrgetfunc	PyObject * PyObject * PyObject *	PyObject *
descrsetfunc	PyObject * PyObject * PyObject *	int
hashfunc	PyObject *	Py_hash_t
richcmpfunc		PyObject *

See *Slot Type typedefs* below for more detail.

12.3.2 PyTypeObject Definition

The structure definition for `PyTypeObject` can be found in `Include/object.h`. For convenience of reference, this repeats the definition found there:

```
typedef struct _typeobject {
    PyObject_VAR_HEAD
    const char *tp_name; /* For printing, in format "<module>.<name>" */
    Py_ssize_t tp_basicsize, tp_itemsize; /* For allocation */

    /* Methods to implement standard operations */

    destructor tp_dealloc;
    Py_ssize_t tp_vectorcall_offset;
    getattrfunc tp_getattr;
    setattrfunc tp_setattr;
    PyAsyncMethods *tp_as_async; /* formerly known as tp_compare (Python 2)
                                    or tp_reserved (Python 3) */
    reprfunc tp_repr;

    /* Method suites for standard classes */

    PyNumberMethods *tp_as_number;
    PySequenceMethods *tp_as_sequence;
    PyMappingMethods *tp_as_mapping;

    /* More standard operations (here for binary compatibility) */

    hashfunc tp_hash;
    ternaryfunc tp_call;
    reprfunc tp_str;
    getattrofunc tp_getattro;
    setattrofunc tp_setattro;

    /* Functions to access object as input/output buffer */
    PyBufferProcs *tp_as_buffer;

    /* Flags to define presence of optional/expanded features */
    unsigned long tp_flags;

    const char *tp_doc; /* Documentation string */

    /* Assigned meaning in release 2.0 */
    /* call function for all accessible objects */
    traverseproc tp_traverse;

    /* delete references to contained objects */
    inquiry tp_clear;

    /* Assigned meaning in release 2.1 */
    /* rich comparisons */
    richcmpfunc tp_richcompare;

    /* weak reference enabler */
    Py_ssize_t tp_weaklistoffset;
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
    /* Iterators */
    getiterfunc tp_iter;
    iternextfunc tp_iternext;

    /* Attribute descriptor and subclassing stuff */
    struct PyMethodDef *tp_methods;
    struct PyMemberDef *tp_members;
    struct PyGetSetDef *tp_getset;
    // Strong reference on a heap type, borrowed reference on a static type
    struct _typeobject *tp_base;
    PyObject *tp_dict;
    descrgetfunc tp_descr_get;
    descrsetfunc tp_descr_set;
    Py_ssize_t tp_dictoffset;
    initproc tp_init;
    allocfunc tp_alloc;
    newfunc tp_new;
    freefunc tp_free; /* Low-level free-memory routine */
    inquiry tp_is_gc; /* For PyObject_IS_GC */
    PyObject *tp_bases;
    PyObject *tp_mro; /* method resolution order */
    PyObject *tp_cache;
    PyObject *tp_subclasses;
    PyObject *tp_weaklist;
    destructor tp_del;

    /* Type attribute cache version tag. Added in version 2.6 */
    unsigned int tp_version_tag;

    destructor tp_finalize;
    vectorcallfunc tp_vectorcall;

    /* bitset of which type-watchers care about this type */
    unsigned char tp_watched;
} PyTypeObject;
```

12.3.3 PyObject Slots

The type object structure extends the `PyVarObject` structure. The `ob_size` field is used for dynamic types (created by `type_new()`, usually called from a class statement). Note that `PyType_Type` (the metatype) initializes `tp_itemsize`, which means that its instances (i.e. type objects) *must* have the `ob_size` field.

Py_ssize_t `PyObject`**.ob_refcnt**

Część stabilnego ABI. This is the type object's reference count, initialized to `1` by the `PyObject_HEAD_INIT` macro. Note that for *statically allocated type objects*, the type's instances (objects whose `ob_type` points back to the type) do *not* count as references. But for *dynamically allocated type objects*, the instances *do* count as references.

Inheritance:

This field is not inherited by subtypes.

PyTypeObject *`PyObject`**.ob_type**

Część stabilnego ABI. This is the type's type, in other words its metatype. It is initialized by the argument to the `PyObject_HEAD_INIT` macro, and its value should normally be `&PyType_Type`. However, for dynamically loadable extension modules that must be usable on Windows (at least), the compiler complains that this is not a valid initializer. Therefore, the convention is to pass `NULL` to the `PyObject_HEAD_INIT` macro and to

initialize this field explicitly at the start of the module's initialization function, before doing anything else. This is typically done like this:

```
Foo_Type.ob_type = &PyType_Type;
```

This should be done before any instances of the type are created. `PyType_Ready()` checks if `ob_type` is `NULL`, and if so, initializes it to the `ob_type` field of the base class. `PyType_Ready()` will not change this field if it is non-zero.

Inheritance:

This field is inherited by subtypes.

12.3.4 PyVarObject Slots

Py_ssize_t `PyVarObject.`**`ob_size`**

Część stabilnego ABI. For *statically allocated type objects*, this should be initialized to zero. For *dynamically allocated type objects*, this field has a special internal meaning.

Inheritance:

This field is not inherited by subtypes.

12.3.5 PyTypeObject Slots

Each slot has a section describing inheritance. If `PyType_Ready()` may set a value when the field is set to `NULL` then there will also be a „Default" section. (Note that many fields set on `PyBaseObject_Type` and `PyType_Type` effectively act as defaults.)

const char *`PyTypeObject.`**`tp_name`**

Pointer to a NUL-terminated string containing the name of the type. For types that are accessible as module globals, the string should be the full module name, followed by a dot, followed by the type name; for built-in types, it should be just the type name. If the module is a submodule of a package, the full package name is part of the full module name. For example, a type named `T` defined in module `M` in subpackage `Q` in package `P` should have the `tp_name` initializer `"P.Q.M.T"`.

For *dynamically allocated type objects*, this should just be the type name, and the module name explicitly stored in the type dict as the value for key `'__module__'`.

For *statically allocated type objects*, the *tp_name* field should contain a dot. Everything before the last dot is made accessible as the `__module__` attribute, and everything after the last dot is made accessible as the `__name__` attribute.

If no dot is present, the entire `tp_name` field is made accessible as the `__name__` attribute, and the `__module__` attribute is undefined (unless explicitly set in the dictionary, as explained above). This means your type will be impossible to pickle. Additionally, it will not be listed in module documentations created with pydoc.

This field must not be `NULL`. It is the only required field in `PyTypeObject()` (other than potentially `tp_itemsize`).

Inheritance:

This field is not inherited by subtypes.

Py_ssize_t `PyTypeObject.`**`tp_basicsize`**

Py_ssize_t `PyTypeObject.`**`tp_itemsize`**

These fields allow calculating the size in bytes of instances of the type.

There are two kinds of types: types with fixed-length instances have a zero `tp_itemsize` field, types with variable-length instances have a non-zero `tp_itemsize` field. For a type with fixed-length instances, all instances have the same size, given in `tp_basicsize`.

For a type with variable-length instances, the instances must have an `ob_size` field, and the instance size is `tp_basicsize` plus N times `tp_itemsize`, where N is the „length" of the object. The value of N is typically

stored in the instance's *ob_size* field. There are exceptions: for example, ints use a negative *ob_size* to indicate a negative number, and N is `abs(ob_size)` there. Also, the presence of an *ob_size* field in the instance layout doesn't mean that the instance structure is variable-length (for example, the structure for the list type has fixed-length instances, yet those instances have a meaningful *ob_size* field).

The basic size includes the fields in the instance declared by the macro *PyObject_HEAD* or *PyObject_VAR_HEAD* (whichever is used to declare the instance struct) and this in turn includes the `_ob_prev` and `_ob_next` fields if they are present. This means that the only correct way to get an initializer for the *tp_basicsize* is to use the `sizeof` operator on the struct used to declare the instance layout. The basic size does not include the GC header size.

A note about alignment: if the variable items require a particular alignment, this should be taken care of by the value of *tp_basicsize*. Example: suppose a type implements an array of `double`. *tp_itemsize* is `sizeof(double)`. It is the programmer's responsibility that *tp_basicsize* is a multiple of `sizeof(double)` (assuming this is the alignment requirement for `double`).

For any type with variable-length instances, this field must not be `NULL`.

Inheritance:

These fields are inherited separately by subtypes. If the base type has a non-zero *tp_itemsize*, it is generally not safe to set *tp_itemsize* to a different non-zero value in a subtype (though this depends on the implementation of the base type).

destructor *PyTypeObject*.**tp_dealloc**

A pointer to the instance destructor function. This function must be defined unless the type guarantees that its instances will never be deallocated (as is the case for the singletons `None` and `Ellipsis`). The function signature is:

```
void tp_dealloc(PyObject *self);
```

The destructor function is called by the *Py_DECREF()* and *Py_XDECREF()* macros when the new reference count is zero. At this point, the instance is still in existence, but there are no references to it. The destructor function should free all references which the instance owns, free all memory buffers owned by the instance (using the freeing function corresponding to the allocation function used to allocate the buffer), and call the type's *tp_free* function. If the type is not subtypable (doesn't have the *Py_TPFLAGS_BASETYPE* flag bit set), it is permissible to call the object deallocator directly instead of via *tp_free*. The object deallocator should be the one used to allocate the instance; this is normally *PyObject_Del()* if the instance was allocated using *PyObject_New* or *PyObject_NewVar*, or *PyObject_GC_Del()* if the instance was allocated using *PyObject_GC_New* or *PyObject_GC_NewVar*.

If the type supports garbage collection (has the *Py_TPFLAGS_HAVE_GC* flag bit set), the destructor should call *PyObject_GC_UnTrack()* before clearing any member fields.

```
static void foo_dealloc(foo_object *self) {
    PyObject_GC_UnTrack(self);
    Py_CLEAR(self->ref);
    Py_TYPE(self)->tp_free((PyObject *)self);
}
```

Finally, if the type is heap allocated (*Py_TPFLAGS_HEAPTYPE*), the deallocator should release the owned reference to its type object (via *Py_DECREF()*) after calling the type deallocator. In order to avoid dangling pointers, the recommended way to achieve this is:

```
static void foo_dealloc(foo_object *self) {
    PyTypeObject *tp = Py_TYPE(self);
    // free references and buffers here
    tp->tp_free(self);
    Py_DECREF(tp);
}
```

Inheritance:

This field is inherited by subtypes.

Py_ssize_t `PyTypeObject`.**tp_vectorcall_offset**

An optional offset to a per-instance function that implements calling the object using the *vectorcall protocol*, a more efficient alternative of the simpler `tp_call`.

This field is only used if the flag `Py_TPFLAGS_HAVE_VECTORCALL` is set. If so, this must be a positive integer containing the offset in the instance of a `vectorcallfunc` pointer.

The *vectorcallfunc* pointer may be `NULL`, in which case the instance behaves as if `Py_TPFLAGS_HAVE_VECTORCALL` was not set: calling the instance falls back to `tp_call`.

Any class that sets `Py_TPFLAGS_HAVE_VECTORCALL` must also set `tp_call` and make sure its behaviour is consistent with the *vectorcallfunc* function. This can be done by setting *tp_call* to `PyVectorcall_Call()`.

Zmienione w wersji 3.8: Before version 3.8, this slot was named `tp_print`. In Python 2.x, it was used for printing to a file. In Python 3.0 to 3.7, it was unused.

Zmienione w wersji 3.12: Before version 3.12, it was not recommended for *mutable heap types* to implement the vectorcall protocol. When a user sets `__call__` in Python code, only *tp_call* is updated, likely making it inconsistent with the vectorcall function. Since 3.12, setting `__call__` will disable vectorcall optimization by clearing the `Py_TPFLAGS_HAVE_VECTORCALL` flag.

Inheritance:

This field is always inherited. However, the `Py_TPFLAGS_HAVE_VECTORCALL` flag is not always inherited. If it's not set, then the subclass won't use *vectorcall*, except when `PyVectorcall_Call()` is explicitly called.

getattrfunc `PyTypeObject`.**tp_getattr**

An optional pointer to the get-attribute-string function.

This field is deprecated. When it is defined, it should point to a function that acts the same as the `tp_getattro` function, but taking a C string instead of a Python string object to give the attribute name.

Inheritance:

Group: `tp_getattr`, `tp_getattro`

This field is inherited by subtypes together with `tp_getattro`: a subtype inherits both `tp_getattr` and `tp_getattro` from its base type when the subtype's `tp_getattr` and `tp_getattro` are both `NULL`.

setattrfunc `PyTypeObject`.**tp_setattr**

An optional pointer to the function for setting and deleting attributes.

This field is deprecated. When it is defined, it should point to a function that acts the same as the `tp_setattro` function, but taking a C string instead of a Python string object to give the attribute name.

Inheritance:

Group: `tp_setattr`, `tp_setattro`

This field is inherited by subtypes together with `tp_setattro`: a subtype inherits both `tp_setattr` and `tp_setattro` from its base type when the subtype's `tp_setattr` and `tp_setattro` are both `NULL`.

PyAsyncMethods **`PyTypeObject`.**tp_as_async**

Pointer to an additional structure that contains fields relevant only to objects which implement *awaitable* and *asynchronous iterator* protocols at the C-level. See *Async Object Structures* for details.

Added in version 3.5: Formerly known as `tp_compare` and `tp_reserved`.

Inheritance:

The `tp_as_async` field is not inherited, but the contained fields are inherited individually.

reprfunc `PyTypeObject`.**tp_repr**

An optional pointer to a function that implements the built-in function `repr()`.

The signature is the same as for `PyObject_Repr()`:

```
PyObject *tp_repr(PyObject *self);
```

The function must return a string or a Unicode object. Ideally, this function should return a string that, when passed to `eval()`, given a suitable environment, returns an object with the same value. If this is not feasible, it should return a string starting with `'<'` and ending with `'>'` from which both the type and the value of the object can be deduced.

Inheritance:

This field is inherited by subtypes.

Default:

When this field is not set, a string of the form `<%s object at %p>` is returned, where `%s` is replaced by the type name, and `%p` by the object's memory address.

PyNumberMethods **PyTypeObject*.**tp_as_number**

Pointer to an additional structure that contains fields relevant only to objects which implement the number protocol. These fields are documented in *Number Object Structures*.

Inheritance:

The `tp_as_number` field is not inherited, but the contained fields are inherited individually.

PySequenceMethods **PyTypeObject*.**tp_as_sequence**

Pointer to an additional structure that contains fields relevant only to objects which implement the sequence protocol. These fields are documented in *Sequence Object Structures*.

Inheritance:

The `tp_as_sequence` field is not inherited, but the contained fields are inherited individually.

PyMappingMethods **PyTypeObject*.**tp_as_mapping**

Pointer to an additional structure that contains fields relevant only to objects which implement the mapping protocol. These fields are documented in *Mapping Object Structures*.

Inheritance:

The `tp_as_mapping` field is not inherited, but the contained fields are inherited individually.

hashfunc `PyTypeObject`.**tp_hash**

An optional pointer to a function that implements the built-in function `hash()`.

The signature is the same as for `PyObject_Hash()`:

```
Py_hash_t tp_hash(PyObject *);
```

The value `-1` should not be returned as a normal return value; when an error occurs during the computation of the hash value, the function should set an exception and return `-1`.

When this field is not set (*and* `tp_richcompare` is not set), an attempt to take the hash of the object raises `TypeError`. This is the same as setting it to `PyObject_HashNotImplemented()`.

This field can be set explicitly to `PyObject_HashNotImplemented()` to block inheritance of the hash method from a parent type. This is interpreted as the equivalent of `__hash__ = None` at the Python level, causing `isinstance(o, collections.Hashable)` to correctly return `False`. Note that the converse is also true - setting `__hash__ = None` on a class at the Python level will result in the `tp_hash` slot being set to `PyObject_HashNotImplemented()`.

Inheritance:

Group: `tp_hash`, `tp_richcompare`

This field is inherited by subtypes together with `tp_richcompare`: a subtype inherits both of `tp_richcompare` and `tp_hash`, when the subtype's `tp_richcompare` and `tp_hash` are both `NULL`.

Default:

PyBaseObject_Type uses *PyObject_GenericHash()*.

ternaryfunc *PyTypeObject*.**tp_call**

An optional pointer to a function that implements calling the object. This should be NULL if the object is not callable. The signature is the same as for *PyObject_Call()*:

```
PyObject *tp_call(PyObject *self, PyObject *args, PyObject *kwargs);
```

Inheritance:

This field is inherited by subtypes.

reprfunc *PyTypeObject*.**tp_str**

An optional pointer to a function that implements the built-in operation str(). (Note that str is a type now, and str() calls the constructor for that type. This constructor calls *PyObject_Str()* to do the actual work, and *PyObject_Str()* will call this handler.)

The signature is the same as for *PyObject_Str()*:

```
PyObject *tp_str(PyObject *self);
```

The function must return a string or a Unicode object. It should be a „friendly" string representation of the object, as this is the representation that will be used, among other things, by the print() function.

Inheritance:

This field is inherited by subtypes.

Default:

When this field is not set, *PyObject_Repr()* is called to return a string representation.

getattrofunc *PyTypeObject*.**tp_getattro**

An optional pointer to the get-attribute function.

The signature is the same as for *PyObject_GetAttr()*:

```
PyObject *tp_getattro(PyObject *self, PyObject *attr);
```

It is usually convenient to set this field to *PyObject_GenericGetAttr()*, which implements the normal way of looking for object attributes.

Inheritance:

Group: *tp_getattr*, *tp_getattro*

This field is inherited by subtypes together with *tp_getattr*: a subtype inherits both *tp_getattr* and *tp_getattro* from its base type when the subtype's *tp_getattr* and *tp_getattro* are both NULL.

Default:

PyBaseObject_Type uses *PyObject_GenericGetAttr()*.

setattrofunc *PyTypeObject*.**tp_setattro**

An optional pointer to the function for setting and deleting attributes.

The signature is the same as for *PyObject_SetAttr()*:

```
int tp_setattro(PyObject *self, PyObject *attr, PyObject *value);
```

In addition, setting *value* to NULL to delete an attribute must be supported. It is usually convenient to set this field to *PyObject_GenericSetAttr()*, which implements the normal way of setting object attributes.

Inheritance:

Group: *tp_setattr*, *tp_setattro*

This field is inherited by subtypes together with `tp_setattr`: a subtype inherits both `tp_setattr` and `tp_setattro` from its base type when the subtype's `tp_setattr` and `tp_setattro` are both `NULL`.

Default:

`PyBaseObject_Type` uses `PyObject_GenericSetAttr()`.

PyBufferProcs *`PyTypeObject`.**tp_as_buffer**

Pointer to an additional structure that contains fields relevant only to objects which implement the buffer interface. These fields are documented in *Buffer Object Structures*.

Inheritance:

The `tp_as_buffer` field is not inherited, but the contained fields are inherited individually.

unsigned long `PyTypeObject`.**tp_flags**

This field is a bit mask of various flags. Some flags indicate variant semantics for certain situations; others are used to indicate that certain fields in the type object (or in the extension structures referenced via `tp_as_number`, `tp_as_sequence`, `tp_as_mapping`, and `tp_as_buffer`) that were historically not always present are valid; if such a flag bit is clear, the type fields it guards must not be accessed and must be considered to have a zero or `NULL` value instead.

Inheritance:

Inheritance of this field is complicated. Most flag bits are inherited individually, i.e. if the base type has a flag bit set, the subtype inherits this flag bit. The flag bits that pertain to extension structures are strictly inherited if the extension structure is inherited, i.e. the base type's value of the flag bit is copied into the subtype together with a pointer to the extension structure. The `Py_TPFLAGS_HAVE_GC` flag bit is inherited together with the `tp_traverse` and `tp_clear` fields, i.e. if the `Py_TPFLAGS_HAVE_GC` flag bit is clear in the subtype and the `tp_traverse` and `tp_clear` fields in the subtype exist and have `NULL` values. .. XXX are most flag bits *really* inherited individually?

Default:

`PyBaseObject_Type` uses `Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE`.

Bit Masks:

The following bit masks are currently defined; these can be ORed together using the `|` operator to form the value of the `tp_flags` field. The macro `PyType_HasFeature()` takes a type and a flags value, *tp* and *f*, and checks whether `tp->tp_flags & f` is non-zero.

Py_TPFLAGS_HEAPTYPE

This bit is set when the type object itself is allocated on the heap, for example, types created dynamically using `PyType_FromSpec()`. In this case, the `ob_type` field of its instances is considered a reference to the type, and the type object is INCREF'ed when a new instance is created, and DECREF'ed when an instance is destroyed (this does not apply to instances of subtypes; only the type referenced by the instance's ob_type gets INCREF'ed or DECREF'ed). Heap types should also *support garbage collection* as they can form a reference cycle with their own module object.

Inheritance:

???

Py_TPFLAGS_BASETYPE

This bit is set when the type can be used as the base type of another type. If this bit is clear, the type cannot be subtyped (similar to a „final" class in Java).

Inheritance:

???

Py_TPFLAGS_READY

This bit is set when the type object has been fully initialized by `PyType_Ready()`.

Inheritance:

???

Py_TPFLAGS_READYING

This bit is set while `PyType_Ready()` is in the process of initializing the type object.

Inheritance:

???

Py_TPFLAGS_HAVE_GC

This bit is set when the object supports garbage collection. If this bit is set, instances must be created using `PyObject_GC_New` and destroyed using `PyObject_GC_Del()`. More information in section *Supporting Cyclic Garbage Collection*. This bit also implies that the GC-related fields `tp_traverse` and `tp_clear` are present in the type object.

Inheritance:

Group: `Py_TPFLAGS_HAVE_GC`, `tp_traverse`, `tp_clear`

The `Py_TPFLAGS_HAVE_GC` flag bit is inherited together with the `tp_traverse` and `tp_clear` fields, i.e. if the `Py_TPFLAGS_HAVE_GC` flag bit is clear in the subtype and the `tp_traverse` and `tp_clear` fields in the subtype exist and have `NULL` values.

Py_TPFLAGS_DEFAULT

This is a bitmask of all the bits that pertain to the existence of certain fields in the type object and its extension structures. Currently, it includes the following bits: `Py_TPFLAGS_HAVE_STACKLESS_EXTENSION`.

Inheritance:

???

Py_TPFLAGS_METHOD_DESCRIPTOR

This bit indicates that objects behave like unbound methods.

If this flag is set for `type(meth)`, then:

- `meth.__get__(obj, cls)(*args, **kwds)` (with `obj` not None) must be equivalent to `meth(obj, *args, **kwds)`.
- `meth.__get__(None, cls)(*args, **kwds)` must be equivalent to `meth(*args, **kwds)`.

This flag enables an optimization for typical method calls like `obj.meth()`: it avoids creating a temporary „bound method" object for `obj.meth`.

Added in version 3.8.

Inheritance:

This flag is never inherited by types without the `Py_TPFLAGS_IMMUTABLETYPE` flag set. For extension types, it is inherited whenever `tp_descr_get` is inherited.

Py_TPFLAGS_MANAGED_DICT

This bit indicates that instances of the class have a *~object.__dict__* attribute, and that the space for the dictionary is managed by the VM.

If this flag is set, `Py_TPFLAGS_HAVE_GC` should also be set.

The type traverse function must call `PyObject_VisitManagedDict()` and its clear function must call `PyObject_ClearManagedDict()`.

Added in version 3.12.

Inheritance:

This flag is inherited unless the `tp_dictoffset` field is set in a superclass.

Py_TPFLAGS_MANAGED_WEAKREF

This bit indicates that instances of the class should be weakly referenceable.

Added in version 3.12.

Inheritance:

This flag is inherited unless the *tp_weaklistoffset* field is set in a superclass.

Py_TPFLAGS_ITEMS_AT_END

Only usable with variable-size types, i.e. ones with non-zero *tp_itemsize*.

Indicates that the variable-sized portion of an instance of this type is at the end of the instance's memory area, at an offset of `Py_TYPE(obj)->tp_basicsize` (which may be different in each subclass).

When setting this flag, be sure that all superclasses either use this memory layout, or are not variable-sized. Python does not check this.

Added in version 3.12.

Inheritance:

This flag is inherited.

Py_TPFLAGS_LONG_SUBCLASS

Py_TPFLAGS_LIST_SUBCLASS

Py_TPFLAGS_TUPLE_SUBCLASS

Py_TPFLAGS_BYTES_SUBCLASS

Py_TPFLAGS_UNICODE_SUBCLASS

Py_TPFLAGS_DICT_SUBCLASS

Py_TPFLAGS_BASE_EXC_SUBCLASS

Py_TPFLAGS_TYPE_SUBCLASS

These flags are used by functions such as *PyLong_Check()* to quickly determine if a type is a subclass of a built-in type; such specific checks are faster than a generic check, like *PyObject_IsInstance()*. Custom types that inherit from built-ins should have their *tp_flags* set appropriately, or the code that interacts with such types will behave differently depending on what kind of check is used.

Py_TPFLAGS_HAVE_FINALIZE

This bit is set when the *tp_finalize* slot is present in the type structure.

Added in version 3.4.

Niezalecane od wersji 3.8: This flag isn't necessary anymore, as the interpreter assumes the *tp_finalize* slot is always present in the type structure.

Py_TPFLAGS_HAVE_VECTORCALL

This bit is set when the class implements the *vectorcall protocol*. See *tp_vectorcall_offset* for details.

Inheritance:

This bit is inherited if *tp_call* is also inherited.

Added in version 3.9.

Zmienione w wersji 3.12: This flag is now removed from a class when the class's `__call__()` method is reassigned.

This flag can now be inherited by mutable classes.

Py_TPFLAGS_IMMUTABLETYPE

This bit is set for type objects that are immutable: type attributes cannot be set nor deleted.

`PyType_Ready()` automatically applies this flag to *static types*.

Inheritance:

This flag is not inherited.

Added in version 3.10.

Py_TPFLAGS_DISALLOW_INSTANTIATION

Disallow creating instances of the type: set `tp_new` to NULL and don't create the `__new__` key in the type dictionary.

The flag must be set before creating the type, not after. For example, it must be set before `PyType_Ready()` is called on the type.

The flag is set automatically on *static types* if `tp_base` is NULL or `&PyBaseObject_Type` and `tp_new` is NULL.

Inheritance:

This flag is not inherited. However, subclasses will not be instantiable unless they provide a non-NULL `tp_new` (which is only possible via the C API).

> **Informacja**
>
> To disallow instantiating a class directly but allow instantiating its subclasses (e.g. for an *abstract base class*), do not use this flag. Instead, make `tp_new` only succeed for subclasses.

Added in version 3.10.

Py_TPFLAGS_MAPPING

This bit indicates that instances of the class may match mapping patterns when used as the subject of a `match` block. It is automatically set when registering or subclassing `collections.abc.Mapping`, and unset when registering `collections.abc.Sequence`.

> **Informacja**
>
> `Py_TPFLAGS_MAPPING` and `Py_TPFLAGS_SEQUENCE` are mutually exclusive; it is an error to enable both flags simultaneously.

Inheritance:

This flag is inherited by types that do not already set `Py_TPFLAGS_SEQUENCE`.

> **Zobacz także**
>
> **PEP 634** – Structural Pattern Matching: Specification

Added in version 3.10.

Py_TPFLAGS_SEQUENCE

This bit indicates that instances of the class may match sequence patterns when used as the subject of a `match` block. It is automatically set when registering or subclassing `collections.abc.Sequence`, and unset when registering `collections.abc.Mapping`.

> **Informacja**
>
> `Py_TPFLAGS_MAPPING` and `Py_TPFLAGS_SEQUENCE` are mutually exclusive; it is an error to enable both flags simultaneously.

Inheritance:

This flag is inherited by types that do not already set `Py_TPFLAGS_MAPPING`.

> **Zobacz także**
>
> **PEP 634** – Structural Pattern Matching: Specification

Added in version 3.10.

Py_TPFLAGS_VALID_VERSION_TAG

Internal. Do not set or unset this flag. To indicate that a class has changed call `PyType_Modified()`

> **Ostrzeżenie**
>
> This flag is present in header files, but is not be used. It will be removed in a future version of CPython

const char *`PyTypeObject`.**tp_doc**

An optional pointer to a NUL-terminated C string giving the docstring for this type object. This is exposed as the `__doc__` attribute on the type and instances of the type.

Inheritance:

This field is *not* inherited by subtypes.

traverseproc `PyTypeObject`.**tp_traverse**

An optional pointer to a traversal function for the garbage collector. This is only used if the `Py_TPFLAGS_HAVE_GC` flag bit is set. The signature is:

```
int tp_traverse(PyObject *self, visitproc visit, void *arg);
```

More information about Python's garbage collection scheme can be found in section *Supporting Cyclic Garbage Collection*.

The `tp_traverse` pointer is used by the garbage collector to detect reference cycles. A typical implementation of a `tp_traverse` function simply calls `Py_VISIT()` on each of the instance's members that are Python objects that the instance owns. For example, this is function `local_traverse()` from the `_thread` extension module:

```
static int
local_traverse(localobject *self, visitproc visit, void *arg)
{
    Py_VISIT(self->args);
    Py_VISIT(self->kw);
    Py_VISIT(self->dict);
    return 0;
}
```

Note that `Py_VISIT()` is called only on those members that can participate in reference cycles. Although there is also a `self->key` member, it can only be `NULL` or a Python string and therefore cannot be part of a reference cycle.

On the other hand, even if you know a member can never be part of a cycle, as a debugging aid you may want to visit it anyway just so the `gc` module's `get_referents()` function will include it.

Heap types (*Py_TPFLAGS_HEAPTYPE*) must visit their type with:

```
Py_VISIT(Py_TYPE(self));
```

It is only needed since Python 3.9. To support Python 3.8 and older, this line must be conditional:

```
#if PY_VERSION_HEX >= 0x03090000
    Py_VISIT(Py_TYPE(self));
#endif
```

If the *Py_TPFLAGS_MANAGED_DICT* bit is set in the *tp_flags* field, the traverse function must call *PyObject_VisitManagedDict()* like this:

```
PyObject_VisitManagedDict((PyObject*)self, visit, arg);
```

Ostrzeżenie

When implementing *tp_traverse*, only the members that the instance *owns* (by having *strong references* to them) must be visited. For instance, if an object supports weak references via the *tp_weaklist* slot, the pointer supporting the linked list (what *tp_weaklist* points to) must **not** be visited as the instance does not directly own the weak references to itself (the weakreference list is there to support the weak reference machinery, but the instance has no strong reference to the elements inside it, as they are allowed to be removed even if the instance is still alive).

Note that *Py_VISIT()* requires the *visit* and *arg* parameters to `local_traverse()` to have these specific names; don't name them just anything.

Instances of *heap-allocated types* hold a reference to their type. Their traversal function must therefore either visit *Py_TYPE(self)*, or delegate this responsibility by calling `tp_traverse` of another heap-allocated type (such as a heap-allocated superclass). If they do not, the type object may not be garbage-collected.

Zmienione w wersji 3.9: Heap-allocated types are expected to visit `Py_TYPE(self)` in `tp_traverse`. In earlier versions of Python, due to bug 40217, doing this may lead to crashes in subclasses.

Inheritance:

Group: *Py_TPFLAGS_HAVE_GC*, *tp_traverse*, *tp_clear*

This field is inherited by subtypes together with *tp_clear* and the *Py_TPFLAGS_HAVE_GC* flag bit: the flag bit, *tp_traverse*, and *tp_clear* are all inherited from the base type if they are all zero in the subtype.

inquiry *PyTypeObject*.**tp_clear**

An optional pointer to a clear function for the garbage collector. This is only used if the *Py_TPFLAGS_HAVE_GC* flag bit is set. The signature is:

```
int tp_clear(PyObject *);
```

The *tp_clear* member function is used to break reference cycles in cyclic garbage detected by the garbage collector. Taken together, all *tp_clear* functions in the system must combine to break all reference cycles. This is subtle, and if in any doubt supply a *tp_clear* function. For example, the tuple type does not implement a *tp_clear* function, because it's possible to prove that no reference cycle can be composed entirely of tuples. Therefore the *tp_clear* functions of other types must be sufficient to break any cycle containing a tuple. This isn't immediately obvious, and there's rarely a good reason to avoid implementing *tp_clear*.

Implementations of *tp_clear* should drop the instance's references to those of its members that may be Python objects, and set its pointers to those members to `NULL`, as in the following example:

```
static int
local_clear(localobject *self)
{
    Py_CLEAR(self->key);
    Py_CLEAR(self->args);
    Py_CLEAR(self->kw);
    Py_CLEAR(self->dict);
    return 0;
}
```

The `Py_CLEAR()` macro should be used, because clearing references is delicate: the reference to the contained object must not be released (via `Py_DECREF()`) until after the pointer to the contained object is set to `NULL`. This is because releasing the reference may cause the contained object to become trash, triggering a chain of reclamation activity that may include invoking arbitrary Python code (due to finalizers, or weakref callbacks, associated with the contained object). If it's possible for such code to reference *self* again, it's important that the pointer to the contained object be `NULL` at that time, so that *self* knows the contained object can no longer be used. The `Py_CLEAR()` macro performs the operations in a safe order.

If the `Py_TPFLAGS_MANAGED_DICT` bit is set in the `tp_flags` field, the traverse function must call `PyObject_ClearManagedDict()` like this:

```
PyObject_ClearManagedDict((PyObject*)self);
```

Note that `tp_clear` is not *always* called before an instance is deallocated. For example, when reference counting is enough to determine that an object is no longer used, the cyclic garbage collector is not involved and `tp_dealloc` is called directly.

Because the goal of `tp_clear` functions is to break reference cycles, it's not necessary to clear contained objects like Python strings or Python integers, which can't participate in reference cycles. On the other hand, it may be convenient to clear all contained Python objects, and write the type's `tp_dealloc` function to invoke `tp_clear`.

More information about Python's garbage collection scheme can be found in section *Supporting Cyclic Garbage Collection*.

Inheritance:

Group: `Py_TPFLAGS_HAVE_GC`, `tp_traverse`, `tp_clear`

This field is inherited by subtypes together with `tp_traverse` and the `Py_TPFLAGS_HAVE_GC` flag bit: the flag bit, `tp_traverse`, and `tp_clear` are all inherited from the base type if they are all zero in the subtype.

richcmpfunc `PyTypeObject.`**`tp_richcompare`**

An optional pointer to the rich comparison function, whose signature is:

```
PyObject *tp_richcompare(PyObject *self, PyObject *other, int op);
```

The first parameter is guaranteed to be an instance of the type that is defined by `PyTypeObject`.

The function should return the result of the comparison (usually `Py_True` or `Py_False`). If the comparison is undefined, it must return `Py_NotImplemented`, if another error occurred it must return `NULL` and set an exception condition.

The following constants are defined to be used as the third argument for `tp_richcompare` and for `PyObject_RichCompare()`:

Stała	Comparison
Py_LT	<
Py_LE	<=
Py_EQ	==
Py_NE	!=
Py_GT	>
Py_GE	>=

The following macro is defined to ease writing rich comparison functions:

Py_RETURN_RICHCOMPARE(VAL_A, VAL_B, op)

Return `Py_True` or `Py_False` from the function, depending on the result of a comparison. VAL_A and VAL_B must be orderable by C comparison operators (for example, they may be C ints or floats). The third argument specifies the requested operation, as for `PyObject_RichCompare()`.

The returned value is a new *strong reference*.

On error, sets an exception and returns `NULL` from the function.

Added in version 3.7.

Inheritance:

Group: `tp_hash`, `tp_richcompare`

This field is inherited by subtypes together with `tp_hash`: a subtype inherits `tp_richcompare` and `tp_hash` when the subtype's `tp_richcompare` and `tp_hash` are both `NULL`.

Default:

`PyBaseObject_Type` provides a `tp_richcompare` implementation, which may be inherited. However, if only `tp_hash` is defined, not even the inherited function is used and instances of the type will not be able to participate in any comparisons.

Py_ssize_t `PyTypeObject`.**tp_weaklistoffset**

While this field is still supported, `Py_TPFLAGS_MANAGED_WEAKREF` should be used instead, if at all possible.

If the instances of this type are weakly referenceable, this field is greater than zero and contains the offset in the instance structure of the weak reference list head (ignoring the GC header, if present); this offset is used by `PyObject_ClearWeakRefs()` and the `PyWeakref_*` functions. The instance structure needs to include a field of type `PyObject*` which is initialized to `NULL`.

Do not confuse this field with `tp_weaklist`; that is the list head for weak references to the type object itself.

It is an error to set both the `Py_TPFLAGS_MANAGED_WEAKREF` bit and `tp_weaklistoffset`.

Inheritance:

This field is inherited by subtypes, but see the rules listed below. A subtype may override this offset; this means that the subtype uses a different weak reference list head than the base type. Since the list head is always found via `tp_weaklistoffset`, this should not be a problem.

Default:

If the `Py_TPFLAGS_MANAGED_WEAKREF` bit is set in the `tp_flags` field, then `tp_weaklistoffset` will be set to a negative value, to indicate that it is unsafe to use this field.

getiterfunc `PyTypeObject.tp_iter`

An optional pointer to a function that returns an *iterator* for the object. Its presence normally signals that the instances of this type are *iterable* (although sequences may be iterable without this function).

This function has the same signature as `PyObject_GetIter()`:

```
PyObject *tp_iter(PyObject *self);
```

Inheritance:

This field is inherited by subtypes.

iternextfunc `PyTypeObject.tp_iternext`

An optional pointer to a function that returns the next item in an *iterator*. The signature is:

```
PyObject *tp_iternext(PyObject *self);
```

When the iterator is exhausted, it must return `NULL`; a `StopIteration` exception may or may not be set. When another error occurs, it must return `NULL` too. Its presence signals that the instances of this type are iterators.

Iterator types should also define the `tp_iter` function, and that function should return the iterator instance itself (not a new iterator instance).

This function has the same signature as `PyIter_Next()`.

Inheritance:

This field is inherited by subtypes.

struct *PyMethodDef* *`PyTypeObject.tp_methods`

An optional pointer to a static `NULL`-terminated array of `PyMethodDef` structures, declaring regular methods of this type.

For each entry in the array, an entry is added to the type's dictionary (see `tp_dict` below) containing a method descriptor.

Inheritance:

This field is not inherited by subtypes (methods are inherited through a different mechanism).

struct *PyMemberDef* *`PyTypeObject.tp_members`

An optional pointer to a static `NULL`-terminated array of `PyMemberDef` structures, declaring regular data members (fields or slots) of instances of this type.

For each entry in the array, an entry is added to the type's dictionary (see `tp_dict` below) containing a member descriptor.

Inheritance:

This field is not inherited by subtypes (members are inherited through a different mechanism).

struct *PyGetSetDef* *`PyTypeObject.tp_getset`

An optional pointer to a static `NULL`-terminated array of `PyGetSetDef` structures, declaring computed attributes of instances of this type.

For each entry in the array, an entry is added to the type's dictionary (see `tp_dict` below) containing a getset descriptor.

Inheritance:

This field is not inherited by subtypes (computed attributes are inherited through a different mechanism).

PyTypeObject *`PyTypeObject`.**tp_base**

An optional pointer to a base type from which type properties are inherited. At this level, only single inheritance is supported; multiple inheritance require dynamically creating a type object by calling the metatype.

> **Informacja**
>
> Slot initialization is subject to the rules of initializing globals. C99 requires the initializers to be „address constants". Function designators like `PyType_GenericNew()`, with implicit conversion to a pointer, are valid C99 address constants.
>
> However, the unary «&» operator applied to a non-static variable like `PyBaseObject_Type` is not required to produce an address constant. Compilers may support this (gcc does), MSVC does not. Both compilers are strictly standard conforming in this particular behavior.
>
> Consequently, `tp_base` should be set in the extension module's init function.

Inheritance:

This field is not inherited by subtypes (obviously).

Default:

This field defaults to `&PyBaseObject_Type` (which to Python programmers is known as the type `object`).

PyObject *`PyTypeObject`.**tp_dict**

The type's dictionary is stored here by `PyType_Ready()`.

This field should normally be initialized to `NULL` before PyType_Ready is called; it may also be initialized to a dictionary containing initial attributes for the type. Once `PyType_Ready()` has initialized the type, extra attributes for the type may be added to this dictionary only if they don't correspond to overloaded operations (like `__add__()`). Once initialization for the type has finished, this field should be treated as read-only.

Some types may not store their dictionary in this slot. Use `PyType_GetDict()` to retrieve the dictionary for an arbitrary type.

Zmienione w wersji 3.12: Internals detail: For static builtin types, this is always `NULL`. Instead, the dict for such types is stored on `PyInterpreterState`. Use `PyType_GetDict()` to get the dict for an arbitrary type.

Inheritance:

This field is not inherited by subtypes (though the attributes defined in here are inherited through a different mechanism).

Default:

If this field is `NULL`, `PyType_Ready()` will assign a new dictionary to it.

> **Ostrzeżenie**
>
> It is not safe to use `PyDict_SetItem()` on or otherwise modify `tp_dict` with the dictionary C-API.

descrgetfunc `PyTypeObject`.**tp_descr_get**

An optional pointer to a „descriptor get" function.

The function signature is:

```
PyObject * tp_descr_get(PyObject *self, PyObject *obj, PyObject *type);
```

Inheritance:

This field is inherited by subtypes.

descrsetfunc `PyTypeObject.tp_descr_set`

An optional pointer to a function for setting and deleting a descriptor's value.

The function signature is:

```
int tp_descr_set(PyObject *self, PyObject *obj, PyObject *value);
```

The *value* argument is set to `NULL` to delete the value.

Inheritance:

This field is inherited by subtypes.

Py_ssize_t `PyTypeObject.tp_dictoffset`

While this field is still supported, `Py_TPFLAGS_MANAGED_DICT` should be used instead, if at all possible.

If the instances of this type have a dictionary containing instance variables, this field is non-zero and contains the offset in the instances of the type of the instance variable dictionary; this offset is used by `PyObject_GenericGetAttr()`.

Do not confuse this field with `tp_dict`; that is the dictionary for attributes of the type object itself.

The value specifies the offset of the dictionary from the start of the instance structure.

The `tp_dictoffset` should be regarded as write-only. To get the pointer to the dictionary call `PyObject_GenericGetDict()`. Calling `PyObject_GenericGetDict()` may need to allocate memory for the dictionary, so it is may be more efficient to call `PyObject_GetAttr()` when accessing an attribute on the object.

It is an error to set both the `Py_TPFLAGS_MANAGED_WEAKREF` bit and `tp_dictoffset`.

Inheritance:

This field is inherited by subtypes. A subtype should not override this offset; doing so could be unsafe, if C code tries to access the dictionary at the previous offset. To properly support inheritance, use `Py_TPFLAGS_MANAGED_DICT`.

Default:

This slot has no default. For *static types*, if the field is `NULL` then no `__dict__` gets created for instances.

If the `Py_TPFLAGS_MANAGED_DICT` bit is set in the `tp_flags` field, then `tp_dictoffset` will be set to `-1`, to indicate that it is unsafe to use this field.

initproc `PyTypeObject.tp_init`

An optional pointer to an instance initialization function.

This function corresponds to the `__init__()` method of classes. Like `__init__()`, it is possible to create an instance without calling `__init__()`, and it is possible to reinitialize an instance by calling its `__init__()` method again.

The function signature is:

```
int tp_init(PyObject *self, PyObject *args, PyObject *kwds);
```

The self argument is the instance to be initialized; the *args* and *kwds* arguments represent positional and keyword arguments of the call to `__init__()`.

The `tp_init` function, if not `NULL`, is called when an instance is created normally by calling its type, after the type's `tp_new` function has returned an instance of the type. If the `tp_new` function returns an instance of some other type that is not a subtype of the original type, no `tp_init` function is called; if `tp_new` returns an instance of a subtype of the original type, the subtype's `tp_init` is called.

Returns `0` on success, `-1` and sets an exception on error.

Inheritance:

This field is inherited by subtypes.

Default:

For *static types* this field does not have a default.

allocfunc `PyTypeObject.tp_alloc`

An optional pointer to an instance allocation function.

The function signature is:

```
PyObject *tp_alloc(PyTypeObject *self, Py_ssize_t nitems);
```

Inheritance:

This field is inherited by static subtypes, but not by dynamic subtypes (subtypes created by a class statement).

Default:

For dynamic subtypes, this field is always set to `PyType_GenericAlloc()`, to force a standard heap allocation strategy.

For static subtypes, `PyBaseObject_Type` uses `PyType_GenericAlloc()`. That is the recommended value for all statically defined types.

newfunc `PyTypeObject.tp_new`

An optional pointer to an instance creation function.

The function signature is:

```
PyObject *tp_new(PyTypeObject *subtype, PyObject *args, PyObject *kwds);
```

The *subtype* argument is the type of the object being created; the *args* and *kwds* arguments represent positional and keyword arguments of the call to the type. Note that *subtype* doesn't have to equal the type whose `tp_new` function is called; it may be a subtype of that type (but not an unrelated type).

The `tp_new` function should call `subtype->tp_alloc(subtype, nitems)` to allocate space for the object, and then do only as much further initialization as is absolutely necessary. Initialization that can safely be ignored or repeated should be placed in the `tp_init` handler. A good rule of thumb is that for immutable types, all initialization should take place in `tp_new`, while for mutable types, most initialization should be deferred to `tp_init`.

Set the `Py_TPFLAGS_DISALLOW_INSTANTIATION` flag to disallow creating instances of the type in Python.

Inheritance:

This field is inherited by subtypes, except it is not inherited by *static types* whose `tp_base` is `NULL` or `&PyBaseObject_Type`.

Default:

For *static types* this field has no default. This means if the slot is defined as `NULL`, the type cannot be called to create new instances; presumably there is some other way to create instances, like a factory function.

freefunc `PyTypeObject.tp_free`

An optional pointer to an instance deallocation function. Its signature is:

```
void tp_free(void *self);
```

An initializer that is compatible with this signature is `PyObject_Free()`.

Inheritance:

This field is inherited by static subtypes, but not by dynamic subtypes (subtypes created by a class statement)

Default:

In dynamic subtypes, this field is set to a deallocator suitable to match `PyType_GenericAlloc()` and the value of the `Py_TPFLAGS_HAVE_GC` flag bit.

For static subtypes, `PyBaseObject_Type` uses `PyObject_Del()`.

inquiry `PyTypeObject`.**tp_is_gc**

An optional pointer to a function called by the garbage collector.

The garbage collector needs to know whether a particular object is collectible or not. Normally, it is sufficient to look at the object's type's `tp_flags` field, and check the `Py_TPFLAGS_HAVE_GC` flag bit. But some types have a mixture of statically and dynamically allocated instances, and the statically allocated instances are not collectible. Such types should define this function; it should return `1` for a collectible instance, and `0` for a non--collectible instance. The signature is:

```
int tp_is_gc(PyObject *self);
```

(The only example of this are types themselves. The metatype, `PyType_Type`, defines this function to distinguish between statically and *dynamically allocated types*.)

Inheritance:

This field is inherited by subtypes.

Default:

This slot has no default. If this field is `NULL`, `Py_TPFLAGS_HAVE_GC` is used as the functional equivalent.

PyObject *`PyTypeObject`.**tp_bases**

Tuple of base types.

This field should be set to `NULL` and treated as read-only. Python will fill it in when the type is `initialized`.

For dynamically created classes, the `Py_tp_bases` `slot` can be used instead of the *bases* argument of `PyType_FromSpecWithBases()`. The argument form is preferred.

> **Ostrzeżenie**
>
> Multiple inheritance does not work well for statically defined types. If you set `tp_bases` to a tuple, Python will not raise an error, but some slots will only be inherited from the first base.

Inheritance:

This field is not inherited.

PyObject *`PyTypeObject`.**tp_mro**

Tuple containing the expanded set of base types, starting with the type itself and ending with `object`, in Method Resolution Order.

This field should be set to `NULL` and treated as read-only. Python will fill it in when the type is `initialized`.

Inheritance:

This field is not inherited; it is calculated fresh by `PyType_Ready()`.

PyObject *`PyTypeObject`.**tp_cache**

Unused. Internal use only.

Inheritance:

This field is not inherited.

void *`PyTypeObject`.**tp_subclasses**

A collection of subclasses. Internal use only. May be an invalid pointer.

To get a list of subclasses, call the Python method `__subclasses__()`.

Zmienione w wersji 3.12: For some types, this field does not hold a valid `PyObject*`. The type was changed to `void*` to indicate this.

Inheritance:

This field is not inherited.

PyObject **PyTypeObject.tp_weaklist*

Weak reference list head, for weak references to this type object. Not inherited. Internal use only.

Zmienione w wersji 3.12: Internals detail: For the static builtin types this is always `NULL`, even if weakrefs are added. Instead, the weakrefs for each are stored on `PyInterpreterState`. Use the public C-API or the internal `_PyObject_GET_WEAKREFS_LISTPTR()` macro to avoid the distinction.

Inheritance:

This field is not inherited.

destructor `PyTypeObject.tp_del`

This field is deprecated. Use `tp_finalize` instead.

unsigned int `PyTypeObject.tp_version_tag`

Used to index into the method cache. Internal use only.

Inheritance:

This field is not inherited.

destructor `PyTypeObject.tp_finalize`

An optional pointer to an instance finalization function. Its signature is:

```
void tp_finalize(PyObject *self);
```

If `tp_finalize` is set, the interpreter calls it once when finalizing an instance. It is called either from the garbage collector (if the instance is part of an isolated reference cycle) or just before the object is deallocated. Either way, it is guaranteed to be called before attempting to break reference cycles, ensuring that it finds the object in a sane state.

`tp_finalize` should not mutate the current exception status; therefore, a recommended way to write a non--trivial finalizer is:

```
static void
local_finalize(PyObject *self)
{
    PyObject *error_type, *error_value, *error_traceback;

    /* Save the current exception, if any. */
    PyErr_Fetch(&error_type, &error_value, &error_traceback);

    /* ... */

    /* Restore the saved exception. */
    PyErr_Restore(error_type, error_value, error_traceback);
}
```

Also, note that, in a garbage collected Python, `tp_dealloc` may be called from any Python thread, not just the thread which created the object (if the object becomes part of a refcount cycle, that cycle might be collected by a garbage collection on any thread). This is not a problem for Python API calls, since the thread on which tp_dealloc is called will own the Global Interpreter Lock (GIL). However, if the object being destroyed in turn destroys objects from some other C or C++ library, care should be taken to ensure that destroying those objects on the thread which called tp_dealloc will not violate any assumptions of the library.

Inheritance:

This field is inherited by subtypes.

Added in version 3.4.

Zmienione w wersji 3.8: Before version 3.8 it was necessary to set the `Py_TPFLAGS_HAVE_FINALIZE` flags bit in order for this field to be used. This is no longer required.

> **Zobacz także**
>
> „Safe object finalization” (**PEP 442**)

vectorcallfunc `PyTypeObject.tp_vectorcall`

Vectorcall function to use for calls of this type object. In other words, it is used to implement *vectorcall* for `type.__call__`. If `tp_vectorcall` is `NULL`, the default call implementation using `__new__()` and `__init__()` is used.

Inheritance:

This field is never inherited.

Added in version 3.9: (the field exists since 3.8 but it's only used since 3.9)

unsigned char `PyTypeObject.tp_watched`

Internal. Do not use.

Added in version 3.12.

12.3.6 Static Types

Traditionally, types defined in C code are *static*, that is, a static `PyTypeObject` structure is defined directly in code and initialized using `PyType_Ready()`.

This results in types that are limited relative to types defined in Python:

- Static types are limited to one base, i.e. they cannot use multiple inheritance.
- Static type objects (but not necessarily their instances) are immutable. It is not possible to add or modify the type object's attributes from Python.
- Static type objects are shared across *sub-interpreters*, so they should not include any subinterpreter-specific state.

Also, since `PyTypeObject` is only part of the *Limited API* as an opaque struct, any extension modules using static types must be compiled for a specific Python minor version.

12.3.7 Heap Types

An alternative to *static types* is *heap-allocated types*, or *heap types* for short, which correspond closely to classes created by Python's `class` statement. Heap types have the `Py_TPFLAGS_HEAPTYPE` flag set.

This is done by filling a `PyType_Spec` structure and calling `PyType_FromSpec()`, `PyType_FromSpecWithBases()`, `PyType_FromModuleAndSpec()`, or `PyType_FromMetaclass()`.

12.4 Number Object Structures

type **PyNumberMethods**

This structure holds pointers to the functions which an object uses to implement the number protocol. Each function is used by the function of similar name documented in the *Number Protocol* section.

Here is the structure definition:

```
typedef struct {
     binaryfunc nb_add;
     binaryfunc nb_subtract;
     binaryfunc nb_multiply;
     binaryfunc nb_remainder;
     binaryfunc nb_divmod;
     ternaryfunc nb_power;
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
    unaryfunc nb_negative;
    unaryfunc nb_positive;
    unaryfunc nb_absolute;
    inquiry nb_bool;
    unaryfunc nb_invert;
    binaryfunc nb_lshift;
    binaryfunc nb_rshift;
    binaryfunc nb_and;
    binaryfunc nb_xor;
    binaryfunc nb_or;
    unaryfunc nb_int;
    void *nb_reserved;
    unaryfunc nb_float;

    binaryfunc nb_inplace_add;
    binaryfunc nb_inplace_subtract;
    binaryfunc nb_inplace_multiply;
    binaryfunc nb_inplace_remainder;
    ternaryfunc nb_inplace_power;
    binaryfunc nb_inplace_lshift;
    binaryfunc nb_inplace_rshift;
    binaryfunc nb_inplace_and;
    binaryfunc nb_inplace_xor;
    binaryfunc nb_inplace_or;

    binaryfunc nb_floor_divide;
    binaryfunc nb_true_divide;
    binaryfunc nb_inplace_floor_divide;
    binaryfunc nb_inplace_true_divide;

    unaryfunc nb_index;

    binaryfunc nb_matrix_multiply;
    binaryfunc nb_inplace_matrix_multiply;
} PyNumberMethods;
```

Informacja

Binary and ternary functions must check the type of all their operands, and implement the necessary conversions (at least one of the operands is an instance of the defined type). If the operation is not defined for the given operands, binary and ternary functions must return `Py_NotImplemented`, if another error occurred they must return `NULL` and set an exception.

Informacja

The *`nb_reserved`* field should always be `NULL`. It was previously called `nb_long`, and was renamed in Python 3.0.1.

binaryfunc `PyNumberMethods`.**`nb_add`**

binaryfunc `PyNumberMethods`.**`nb_subtract`**

binaryfunc `PyNumberMethods`.**`nb_multiply`**

binaryfunc `PyNumberMethods`.**nb_remainder**

binaryfunc `PyNumberMethods`.**nb_divmod**

ternaryfunc `PyNumberMethods`.**nb_power**

unaryfunc `PyNumberMethods`.**nb_negative**

unaryfunc `PyNumberMethods`.**nb_positive**

unaryfunc `PyNumberMethods`.**nb_absolute**

inquiry `PyNumberMethods`.**nb_bool**

unaryfunc `PyNumberMethods`.**nb_invert**

binaryfunc `PyNumberMethods`.**nb_lshift**

binaryfunc `PyNumberMethods`.**nb_rshift**

binaryfunc `PyNumberMethods`.**nb_and**

binaryfunc `PyNumberMethods`.**nb_xor**

binaryfunc `PyNumberMethods`.**nb_or**

unaryfunc `PyNumberMethods`.**nb_int**

void *`PyNumberMethods`.**nb_reserved**

unaryfunc `PyNumberMethods`.**nb_float**

binaryfunc `PyNumberMethods`.**nb_inplace_add**

binaryfunc `PyNumberMethods`.**nb_inplace_subtract**

binaryfunc `PyNumberMethods`.**nb_inplace_multiply**

binaryfunc `PyNumberMethods`.**nb_inplace_remainder**

ternaryfunc `PyNumberMethods`.**nb_inplace_power**

binaryfunc `PyNumberMethods`.**nb_inplace_lshift**

binaryfunc `PyNumberMethods`.**nb_inplace_rshift**

binaryfunc `PyNumberMethods`.**nb_inplace_and**

binaryfunc `PyNumberMethods`.**nb_inplace_xor**

binaryfunc `PyNumberMethods`.**nb_inplace_or**

binaryfunc `PyNumberMethods`.**nb_floor_divide**

binaryfunc `PyNumberMethods`.**nb_true_divide**

binaryfunc `PyNumberMethods`.**nb_inplace_floor_divide**

binaryfunc `PyNumberMethods`.**nb_inplace_true_divide**

unaryfunc `PyNumberMethods`.**nb_index**

binaryfunc `PyNumberMethods`.**nb_matrix_multiply**

binaryfunc `PyNumberMethods`.**nb_inplace_matrix_multiply**

12.5 Mapping Object Structures

type **PyMappingMethods**

This structure holds pointers to the functions which an object uses to implement the mapping protocol. It has three members:

lenfunc `PyMappingMethods`.**mp_length**

This function is used by `PyMapping_Size()` and `PyObject_Size()`, and has the same signature. This slot may be set to `NULL` if the object has no defined length.

binaryfunc `PyMappingMethods`.**mp_subscript**

This function is used by `PyObject_GetItem()` and `PySequence_GetSlice()`, and has the same signature as `PyObject_GetItem()`. This slot must be filled for the `PyMapping_Check()` function to return `1`, it can be `NULL` otherwise.

objobjargproc `PyMappingMethods`.**mp_ass_subscript**

This function is used by `PyObject_SetItem()`, `PyObject_DelItem()`, `PySequence_SetSlice()` and `PySequence_DelSlice()`. It has the same signature as `PyObject_SetItem()`, but *v* can also be set to `NULL` to delete an item. If this slot is `NULL`, the object does not support item assignment and deletion.

12.6 Sequence Object Structures

type **PySequenceMethods**

This structure holds pointers to the functions which an object uses to implement the sequence protocol.

lenfunc `PySequenceMethods`.**sq_length**

This function is used by `PySequence_Size()` and `PyObject_Size()`, and has the same signature. It is also used for handling negative indices via the `sq_item` and the `sq_ass_item` slots.

binaryfunc `PySequenceMethods`.**sq_concat**

This function is used by `PySequence_Concat()` and has the same signature. It is also used by the `+` operator, after trying the numeric addition via the `nb_add` slot.

ssizeargfunc `PySequenceMethods`.**sq_repeat**

This function is used by `PySequence_Repeat()` and has the same signature. It is also used by the `*` operator, after trying numeric multiplication via the `nb_multiply` slot.

ssizeargfunc `PySequenceMethods`.**sq_item**

This function is used by `PySequence_GetItem()` and has the same signature. It is also used by `PyObject_GetItem()`, after trying the subscription via the `mp_subscript` slot. This slot must be filled for the `PySequence_Check()` function to return `1`, it can be `NULL` otherwise.

Negative indexes are handled as follows: if the `sq_length` slot is filled, it is called and the sequence length is used to compute a positive index which is passed to `sq_item`. If `sq_length` is `NULL`, the index is passed as is to the function.

ssizeobjargproc `PySequenceMethods`.**sq_ass_item**

This function is used by `PySequence_SetItem()` and has the same signature. It is also used by `PyObject_SetItem()` and `PyObject_DelItem()`, after trying the item assignment and deletion via the `mp_ass_subscript` slot. This slot may be left to `NULL` if the object does not support item assignment and deletion.

objobjproc `PySequenceMethods`.**sq_contains**

This function may be used by `PySequence_Contains()` and has the same signature. This slot may be left to `NULL`, in this case `PySequence_Contains()` simply traverses the sequence until it finds a match.

binaryfunc `PySequenceMethods`.**sq_inplace_concat**

This function is used by `PySequence_InPlaceConcat()` and has the same signature. It should modify its first operand, and return it. This slot may be left to `NULL`, in this case `PySequence_InPlaceConcat()` will

fall back to `PySequence_Concat()`. It is also used by the augmented assignment `+=`, after trying numeric in-place addition via the `nb_inplace_add` slot.

ssizeargfunc `PySequenceMethods.sq_inplace_repeat`

This function is used by `PySequence_InPlaceRepeat()` and has the same signature. It should modify its first operand, and return it. This slot may be left to `NULL`, in this case `PySequence_InPlaceRepeat()` will fall back to `PySequence_Repeat()`. It is also used by the augmented assignment `*=`, after trying numeric in-place multiplication via the `nb_inplace_multiply` slot.

12.7 Buffer Object Structures

type **PyBufferProcs**

This structure holds pointers to the functions required by the *Buffer protocol*. The protocol defines how an exporter object can expose its internal data to consumer objects.

getbufferproc `PyBufferProcs.bf_getbuffer`

The signature of this function is:

```
int (PyObject *exporter, Py_buffer *view, int flags);
```

Handle a request to *exporter* to fill in *view* as specified by *flags*. Except for point (3), an implementation of this function MUST take these steps:

(1) Check if the request can be met. If not, raise `BufferError`, set `view->obj` to `NULL` and return `-1`.

(2) Fill in the requested fields.

(3) Increment an internal counter for the number of exports.

(4) Set `view->obj` to *exporter* and increment `view->obj`.

(5) Return `0`.

If *exporter* is part of a chain or tree of buffer providers, two main schemes can be used:

- Re-export: Each member of the tree acts as the exporting object and sets `view->obj` to a new reference to itself.
- Redirect: The buffer request is redirected to the root object of the tree. Here, `view->obj` will be a new reference to the root object.

The individual fields of *view* are described in section *Buffer structure*, the rules how an exporter must react to specific requests are in section *Buffer request types*.

All memory pointed to in the `Py_buffer` structure belongs to the exporter and must remain valid until there are no consumers left. `format`, `shape`, `strides`, `suboffsets` and `internal` are read-only for the consumer.

`PyBuffer_FillInfo()` provides an easy way of exposing a simple bytes buffer while dealing correctly with all request types.

`PyObject_GetBuffer()` is the interface for the consumer that wraps this function.

releasebufferproc `PyBufferProcs.bf_releasebuffer`

The signature of this function is:

```
void (PyObject *exporter, Py_buffer *view);
```

Handle a request to release the resources of the buffer. If no resources need to be released, `PyBufferProcs.bf_releasebuffer` may be `NULL`. Otherwise, a standard implementation of this function will take these optional steps:

(1) Decrement an internal counter for the number of exports.

(2) If the counter is `0`, free all memory associated with *view*.

The exporter MUST use the `internal` field to keep track of buffer-specific resources. This field is guaranteed to remain constant, while a consumer MAY pass a copy of the original buffer as the *view* argument.

This function MUST NOT decrement `view->obj`, since that is done automatically in `PyBuffer_Release()` (this scheme is useful for breaking reference cycles).

`PyBuffer_Release()` is the interface for the consumer that wraps this function.

12.8 Async Object Structures

Added in version 3.5.

type **PyAsyncMethods**

This structure holds pointers to the functions required to implement *awaitable* and *asynchronous iterator* objects.

Here is the structure definition:

```
typedef struct {
    unaryfunc am_await;
    unaryfunc am_aiter;
    unaryfunc am_anext;
    sendfunc am_send;
} PyAsyncMethods;
```

unaryfunc `PyAsyncMethods`.**am_await**

The signature of this function is:

```
PyObject *am_await(PyObject *self);
```

The returned object must be an *iterator*, i.e. `PyIter_Check()` must return `1` for it.

This slot may be set to `NULL` if an object is not an *awaitable*.

unaryfunc `PyAsyncMethods`.**am_aiter**

The signature of this function is:

```
PyObject *am_aiter(PyObject *self);
```

Must return an *asynchronous iterator* object. See `__anext__()` for details.

This slot may be set to `NULL` if an object does not implement asynchronous iteration protocol.

unaryfunc `PyAsyncMethods`.**am_anext**

The signature of this function is:

```
PyObject *am_anext(PyObject *self);
```

Must return an *awaitable* object. See `__anext__()` for details. This slot may be set to `NULL`.

sendfunc `PyAsyncMethods`.**am_send**

The signature of this function is:

```
PySendResult am_send(PyObject *self, PyObject *arg, PyObject **result);
```

See `PyIter_Send()` for details. This slot may be set to `NULL`.

Added in version 3.10.

12.9 Slot Type typedefs

typedef *PyObject* *(***allocfunc**)(*PyTypeObject* *cls, *Py_ssize_t* nitems)

Część stabilnego ABI. The purpose of this function is to separate memory allocation from memory initialization. It should return a pointer to a block of memory of adequate length for the instance, suitably aligned, and initialized to zeros, but with `ob_refcnt` set to `1` and `ob_type` set to the type argument. If the type's `tp_itemsize` is non-zero, the object's `ob_size` field should be initialized to *nitems* and the length of the allocated memory block should be `tp_basicsize + nitems*tp_itemsize`, rounded up to a multiple of `sizeof(void*)`; otherwise, *nitems* is not used and the length of the block should be `tp_basicsize`.

This function should not do any other instance initialization, not even to allocate additional memory; that should be done by `tp_new`.

typedef void (***destructor**)(*PyObject**)

Część stabilnego ABI.

typedef void (***freefunc**)(void*)

See `tp_free`.

typedef *PyObject* *(***newfunc**)(*PyObject**, *PyObject**, *PyObject**)

Część stabilnego ABI. See `tp_new`.

typedef int (***initproc**)(*PyObject**, *PyObject**, *PyObject**)

Część stabilnego ABI. See `tp_init`.

typedef *PyObject* *(***reprfunc**)(*PyObject**)

Część stabilnego ABI. See `tp_repr`.

typedef *PyObject* *(***getattrfunc**)(*PyObject* *self, char *attr)

Część stabilnego ABI. Return the value of the named attribute for the object.

typedef int (***setattrfunc**)(*PyObject* *self, char *attr, *PyObject* *value)

Część stabilnego ABI. Set the value of the named attribute for the object. The value argument is set to `NULL` to delete the attribute.

typedef *PyObject* *(***getattrofunc**)(*PyObject* *self, *PyObject* *attr)

Część stabilnego ABI. Return the value of the named attribute for the object.

See `tp_getattro`.

typedef int (***setattrofunc**)(*PyObject* *self, *PyObject* *attr, *PyObject* *value)

Część stabilnego ABI. Set the value of the named attribute for the object. The value argument is set to `NULL` to delete the attribute.

See `tp_setattro`.

typedef *PyObject* *(***descrgetfunc**)(*PyObject**, *PyObject**, *PyObject**)

Część stabilnego ABI. See `tp_descr_get`.

typedef int (***descrsetfunc**)(*PyObject**, *PyObject**, *PyObject**)

Część stabilnego ABI. See `tp_descr_set`.

typedef *Py_hash_t* (***hashfunc**)(*PyObject**)

Część stabilnego ABI. See `tp_hash`.

typedef *PyObject* *(***richcmpfunc**)(*PyObject**, *PyObject**, int)

Część stabilnego ABI. See `tp_richcompare`.

typedef *PyObject* *(***getiterfunc**)(*PyObject**)

Część stabilnego ABI. See `tp_iter`.

typedef *PyObject* *(***iternextfunc**)(*PyObject**)

Część stabilnego ABI. See `tp_iternext`.

typedef *Py_ssize_t* (*`lenfunc`)(*PyObject**)

Część stabilnego ABI.

typedef int (*`getbufferproc`)(*PyObject**, *Py_buffer**, int)

Część stabilnego ABI *od wersji 3.12*.

typedef void (*`releasebufferproc`)(*PyObject**, *Py_buffer**)

Część stabilnego ABI *od wersji 3.12*.

typedef *PyObject* *(*`unaryfunc`)(*PyObject**)

Część stabilnego ABI.

typedef *PyObject* *(*`binaryfunc`)(*PyObject**, *PyObject**)

Część stabilnego ABI.

typedef *PySendResult* (*`sendfunc`)(*PyObject**, *PyObject**, *PyObject***)

See `am_send`.

typedef *PyObject* *(*`ternaryfunc`)(*PyObject**, *PyObject**, *PyObject**)

Część stabilnego ABI.

typedef *PyObject* *(*`ssizeargfunc`)(*PyObject**, *Py_ssize_t*)

Część stabilnego ABI.

typedef int (*`ssizeobjargproc`)(*PyObject**, *Py_ssize_t*, *PyObject**)

Część stabilnego ABI.

typedef int (*`objobjproc`)(*PyObject**, *PyObject**)

Część stabilnego ABI.

typedef int (*`objobjargproc`)(*PyObject**, *PyObject**, *PyObject**)

Część stabilnego ABI.

12.10 Przykłady

The following are simple examples of Python type definitions. They include common usage you may encounter. Some demonstrate tricky corner cases. For more examples, practical info, and a tutorial, see defining-new-types and new-types-topics.

A basic *static type*:

```
typedef struct {
    PyObject_HEAD
    const char *data;
} MyObject;

static PyTypeObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyObject",
    .tp_basicsize = sizeof(MyObject),
    .tp_doc = PyDoc_STR("My objects"),
    .tp_new = myobj_new,
    .tp_dealloc = (destructor)myobj_dealloc,
    .tp_repr = (reprfunc)myobj_repr,
};
```

You may also find older code (especially in the CPython code base) with a more verbose initializer:

```
static PyTypeObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    "mymod.MyObject",               /* tp_name */
    sizeof(MyObject),               /* tp_basicsize */
    0,                              /* tp_itemsize */
    (destructor)myobj_dealloc,      /* tp_dealloc */
    0,                              /* tp_vectorcall_offset */
    0,                              /* tp_getattr */
    0,                              /* tp_setattr */
    0,                              /* tp_as_async */
    (reprfunc)myobj_repr,           /* tp_repr */
    0,                              /* tp_as_number */
    0,                              /* tp_as_sequence */
    0,                              /* tp_as_mapping */
    0,                              /* tp_hash */
    0,                              /* tp_call */
    0,                              /* tp_str */
    0,                              /* tp_getattro */
    0,                              /* tp_setattro */
    0,                              /* tp_as_buffer */
    0,                              /* tp_flags */
    PyDoc_STR("My objects"),        /* tp_doc */
    0,                              /* tp_traverse */
    0,                              /* tp_clear */
    0,                              /* tp_richcompare */
    0,                              /* tp_weaklistoffset */
    0,                              /* tp_iter */
    0,                              /* tp_iternext */
    0,                              /* tp_methods */
    0,                              /* tp_members */
    0,                              /* tp_getset */
    0,                              /* tp_base */
    0,                              /* tp_dict */
    0,                              /* tp_descr_get */
    0,                              /* tp_descr_set */
    0,                              /* tp_dictoffset */
    0,                              /* tp_init */
    0,                              /* tp_alloc */
    myobj_new,                      /* tp_new */
};
```

A type that supports weakrefs, instance dicts, and hashing:

```
typedef struct {
    PyObject_HEAD
    const char *data;
} MyObject;

static PyTypeObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyObject",
    .tp_basicsize = sizeof(MyObject),
    .tp_doc = PyDoc_STR("My objects"),
    .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE |
         Py_TPFLAGS_HAVE_GC | Py_TPFLAGS_MANAGED_DICT |
         Py_TPFLAGS_MANAGED_WEAKREF,
    .tp_new = myobj_new,
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
    .tp_traverse = (traverseproc)myobj_traverse,
    .tp_clear = (inquiry)myobj_clear,
    .tp_alloc = PyType_GenericNew,
    .tp_dealloc = (destructor)myobj_dealloc,
    .tp_repr = (reprfunc)myobj_repr,
    .tp_hash = (hashfunc)myobj_hash,
    .tp_richcompare = PyBaseObject_Type.tp_richcompare,
};
```

A str subclass that cannot be subclassed and cannot be called to create instances (e.g. uses a separate factory func) using `Py_TPFLAGS_DISALLOW_INSTANTIATION` flag:

```
typedef struct {
    PyUnicodeObject raw;
    char *extra;
} MyStr;

static PyTypeObject MyStr_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyStr",
    .tp_basicsize = sizeof(MyStr),
    .tp_base = NULL,  // set to &PyUnicode_Type in module init
    .tp_doc = PyDoc_STR("my custom str"),
    .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_DISALLOW_INSTANTIATION,
    .tp_repr = (reprfunc)myobj_repr,
};
```

The simplest *static type* with fixed-length instances:

```
typedef struct {
    PyObject_HEAD
} MyObject;

static PyTypeObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyObject",
};
```

The simplest *static type* with variable-length instances:

```
typedef struct {
    PyObject_VAR_HEAD
    const char *data[1];
} MyObject;

static PyTypeObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyObject",
    .tp_basicsize = sizeof(MyObject) - sizeof(char *),
    .tp_itemsize = sizeof(char *),
};
```

12.11 Supporting Cyclic Garbage Collection

Python's support for detecting and collecting garbage which involves circular references requires support from object types which are „containers" for other objects which may also be containers. Types which do not store references to other objects, or which only store references to atomic types (such as numbers or strings), do not need to provide any explicit support for garbage collection.

To create a container type, the `tp_flags` field of the type object must include the `Py_TPFLAGS_HAVE_GC` and provide an implementation of the `tp_traverse` handler. If instances of the type are mutable, a `tp_clear` implementation must also be provided.

Py_TPFLAGS_HAVE_GC

Objects with a type with this flag set must conform with the rules documented here. For convenience these objects will be referred to as container objects.

Constructors for container types must conform to two rules:

1. The memory for the object must be allocated using `PyObject_GC_New` or `PyObject_GC_NewVar`.
2. Once all the fields which may contain references to other containers are initialized, it must call `PyObject_GC_Track()`.

Similarly, the deallocator for the object must conform to a similar pair of rules:

1. Before fields which refer to other containers are invalidated, `PyObject_GC_UnTrack()` must be called.
2. The object's memory must be deallocated using `PyObject_GC_Del()`.

> **Ostrzeżenie**
>
> If a type adds the Py_TPFLAGS_HAVE_GC, then it *must* implement at least a `tp_traverse` handler or explicitly use one from its subclass or subclasses.
>
> When calling `PyType_Ready()` or some of the APIs that indirectly call it like `PyType_FromSpecWithBases()` or `PyType_FromSpec()` the interpreter will automatically populate the `tp_flags`, `tp_traverse` and `tp_clear` fields if the type inherits from a class that implements the garbage collector protocol and the child class does *not* include the `Py_TPFLAGS_HAVE_GC` flag.

PyObject_GC_New(TYPE, typeobj)

Analogous to `PyObject_New` but for container objects with the `Py_TPFLAGS_HAVE_GC` flag set.

PyObject_GC_NewVar(TYPE, typeobj, size)

Analogous to `PyObject_NewVar` but for container objects with the `Py_TPFLAGS_HAVE_GC` flag set.

PyObject ***PyUnstable_Object_GC_NewWithExtraData**(*PyTypeObject* *type, size_t extra_size)

> To jest *niestabilne API*. Może ulec zmianie bez ostrzeżenia w wydaniach minor.

Analogous to `PyObject_GC_New` but allocates *extra_size* bytes at the end of the object (at offset `tp_basicsize`). The allocated memory is initialized to zeros, except for the `Python object header`.

The extra data will be deallocated with the object, but otherwise it is not managed by Python.

> **Ostrzeżenie**
>
> The function is marked as unstable because the final mechanism for reserving extra data after an instance is not yet decided. For allocating a variable number of fields, prefer using `PyVarObject` and `tp_itemsize` instead.

Added in version 3.12.

PyObject_GC_Resize(TYPE, op, newsize)

Resize an object allocated by `PyObject_NewVar`. Returns the resized object of type `TYPE*` (refers to any C type) or `NULL` on failure.

op must be of type `PyVarObject*` and must not be tracked by the collector yet. *newsize* must be of type `Py_ssize_t`.

void **PyObject_GC_Track**(*PyObject* *op)

Część stabilnego ABI. Adds the object *op* to the set of container objects tracked by the collector. The collector can run at unexpected times so objects must be valid while being tracked. This should be called once all the fields followed by the `tp_traverse` handler become valid, usually near the end of the constructor.

int **PyObject_IS_GC**(*PyObject* *obj)

Returns non-zero if the object implements the garbage collector protocol, otherwise returns 0.

The object cannot be tracked by the garbage collector if this function returns 0.

int **PyObject_GC_IsTracked**(*PyObject* *op)

Część stabilnego ABI *od wersji 3.9*. Returns 1 if the object type of *op* implements the GC protocol and *op* is being currently tracked by the garbage collector and 0 otherwise.

This is analogous to the Python function `gc.is_tracked()`.

Added in version 3.9.

int **PyObject_GC_IsFinalized**(*PyObject* *op)

Część stabilnego ABI *od wersji 3.9*. Returns 1 if the object type of *op* implements the GC protocol and *op* has been already finalized by the garbage collector and 0 otherwise.

This is analogous to the Python function `gc.is_finalized()`.

Added in version 3.9.

void **PyObject_GC_Del**(void *op)

Część stabilnego ABI. Releases memory allocated to an object using `PyObject_GC_New` or `PyObject_GC_NewVar`.

void **PyObject_GC_UnTrack**(void *op)

Część stabilnego ABI. Remove the object *op* from the set of container objects tracked by the collector. Note that `PyObject_GC_Track()` can be called again on this object to add it back to the set of tracked objects. The deallocator (`tp_dealloc` handler) should call this for the object before any of the fields used by the `tp_traverse` handler become invalid.

Zmienione w wersji 3.8: The `_PyObject_GC_TRACK()` and `_PyObject_GC_UNTRACK()` macros have been removed from the public C API.

The `tp_traverse` handler accepts a function parameter of this type:

typedef int (***visitproc**)(*PyObject* *object, void *arg)

Część stabilnego ABI. Type of the visitor function passed to the `tp_traverse` handler. The function should be called with an object to traverse as *object* and the third parameter to the `tp_traverse` handler as *arg*. The Python core uses several visitor functions to implement cyclic garbage detection; it's not expected that users will need to write their own visitor functions.

The `tp_traverse` handler must have the following type:

typedef int (***traverseproc**)(*PyObject* *self, *visitproc* visit, void *arg)

Część stabilnego ABI. Traversal function for a container object. Implementations must call the *visit* function for each object directly contained by *self*, with the parameters to *visit* being the contained object and the *arg* value passed to the handler. The *visit* function must not be called with a `NULL` object argument. If *visit* returns a non-zero value that value should be returned immediately.

To simplify writing `tp_traverse` handlers, a `Py_VISIT()` macro is provided. In order to use this macro, the `tp_traverse` implementation must name its arguments exactly *visit* and *arg*:

void **Py_VISIT**(*PyObject* *o)

If *o* is not NULL, call the *visit* callback, with arguments *o* and *arg*. If *visit* returns a non-zero value, then return it. Using this macro, *tp_traverse* handlers look like:

```
static int
my_traverse(Noddy *self, visitproc visit, void *arg)
{
    Py_VISIT(self->foo);
    Py_VISIT(self->bar);
    return 0;
}
```

The *tp_clear* handler must be of the *inquiry* type, or NULL if the object is immutable.

typedef int (***inquiry**)(*PyObject* *self)

Część stabilnego ABI. Drop references that may have created reference cycles. Immutable objects do not have to define this method since they can never directly create reference cycles. Note that the object must still be valid after calling this method (don't just call *Py_DECREF()* on a reference). The collector will call this method if it detects that this object is involved in a reference cycle.

12.11.1 Controlling the Garbage Collector State

The C-API provides the following functions for controlling garbage collection runs.

Py_ssize_t **PyGC_Collect**(void)

Część stabilnego ABI. Perform a full garbage collection, if the garbage collector is enabled. (Note that `gc.collect()` runs it unconditionally.)

Returns the number of collected + unreachable objects which cannot be collected. If the garbage collector is disabled or already collecting, returns `0` immediately. Errors during garbage collection are passed to `sys.unraisablehook`. This function does not raise exceptions.

int **PyGC_Enable**(void)

Część stabilnego ABI *od wersji 3.10.* Enable the garbage collector: similar to `gc.enable()`. Returns the previous state, 0 for disabled and 1 for enabled.

Added in version 3.10.

int **PyGC_Disable**(void)

Część stabilnego ABI *od wersji 3.10.* Disable the garbage collector: similar to `gc.disable()`. Returns the previous state, 0 for disabled and 1 for enabled.

Added in version 3.10.

int **PyGC_IsEnabled**(void)

Część stabilnego ABI *od wersji 3.10.* Query the state of the garbage collector: similar to `gc.isenabled()`. Returns the current state, 0 for disabled and 1 for enabled.

Added in version 3.10.

12.11.2 Querying Garbage Collector State

The C-API provides the following interface for querying information about the garbage collector.

void **PyUnstable_GC_VisitObjects**(*gcvisitobjects_t* callback, void *arg)

> To jest *niestabilne API*. Może ulec zmianie bez ostrzeżenia w wydaniach minor.

Run supplied *callback* on all live GC-capable objects. *arg* is passed through to all invocations of *callback*.

> **Ostrzeżenie**
>
> If new objects are (de)allocated by the callback it is undefined if they will be visited.
>
> Garbage collection is disabled during operation. Explicitly running a collection in the callback may lead to undefined behaviour e.g. visiting the same objects multiple times or not at all.

Added in version 3.12.

typedef int (*`gcvisitobjects_t`)(*PyObject* *object, void *arg)

Type of the visitor function to be passed to `PyUnstable_GC_VisitObjects()`. *arg* is the same as the *arg* passed to `PyUnstable_GC_VisitObjects`. Return `0` to continue iteration, return `1` to stop iteration. Other return values are reserved for now so behavior on returning anything else is undefined.

Added in version 3.12.

ROZDZIAŁ 13

API i wersjonowanie ABI

CPython (standardowa implementacja) ujawnia swój numer wersji w następujących makrach. Należy pamiętać, że odpowiadają one kodowi wersji, z którym został **zbudowany**, niekoniecznie wersji używanej w **czasie działania**.

See *C API Stability* for a discussion of API and ABI stability across versions.

PY_MAJOR_VERSION

`3` w `3.4.1a2`.

PY_MINOR_VERSION

`4` w `3.4.1a2`.

PY_MICRO_VERSION

`1` w `3.4.1a2`.

PY_RELEASE_LEVEL

`a` w `3.4.1a2`. Może to być `0xA` dla wersji alfa, `0xB` dla wersji beta, `0xC` dla wersji kandydującej do wydania lub `0xF` dla wersji finalnej.

PY_RELEASE_SERIAL

`2` w `3.4.1a2`. Zero dla ostatecznych wydań.

PY_VERSION_HEX

Numer wersji Pythona zakodowany w pojedynczej liczbie całkowitej.

Podstawowe informacje wersji można znaleźć, traktując je jako 32-bitową liczbę w następujący sposób:

Bajty	Bity (kolejność big endian)	Znaczenie	Wartość dla `3.4.1a2`
1	1-8	`PY_MAJOR_VERSION`	`0x03`
2	9-16	`PY_MINOR_VERSION`	`0x04`
3	17-24	`PY_MICRO_VERSION`	`0x01`
4	25-28	`PY_RELEASE_LEVEL`	`0xA`
	29-32	`PY_RELEASE_SERIAL`	`0x2`

Tak więc `3.4.1a2` jest hekswersją `0x030401a2`, a `3.10.0` jest hekswersją `0x030a00f0`.

Służy do porównywania liczb, np. `#if PY_VERSION_HEX >= ...`.

Ta strona wersji jest również dostępna za pośrednictwem symbolu `Py_Version`.

const unsigned long **Py_Version**

Część stabilnego ABI *od wersji 3.11.* The Python runtime version number encoded in a single constant integer, with the same format as the `PY_VERSION_HEX` macro. This contains the Python version used at run time.

Added in version 3.11.

Wszystkie podane makra są zdefiniowane w Include/patchlevel.h.

ROZDZIAŁ 14

Monitoring C API

Added in version 3.13.

An extension may need to interact with the event monitoring system. Subscribing to events and registering callbacks can be done via the Python API exposed in `sys.monitoring`.

ROZDZIAŁ 15

Generating Execution Events

The functions below make it possible for an extension to fire monitoring events as it emulates the execution of Python code. Each of these functions accepts a `PyMonitoringState` struct which contains concise information about the activation state of events, as well as the event arguments, which include a `PyObject*` representing the code object, the instruction offset and sometimes additional, event-specific arguments (see `sys.monitoring` for details about the signatures of the different event callbacks). The `codelike` argument should be an instance of `types.CodeType` or of a type that emulates it.

The VM disables tracing when firing an event, so there is no need for user code to do that.

Monitoring functions should not be called with an exception set, except those listed below as working with the current exception.

type **PyMonitoringState**

Representation of the state of an event type. It is allocated by the user while its contents are maintained by the monitoring API functions described below.

All of the functions below return 0 on success and -1 (with an exception set) on error.

See `sys.monitoring` for descriptions of the events.

int **PyMonitoring_FirePyStartEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset)

Fire a `PY_START` event.

int **PyMonitoring_FirePyResumeEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset)

Fire a `PY_RESUME` event.

int **PyMonitoring_FirePyReturnEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, *PyObject* *retval)

Fire a `PY_RETURN` event.

int **PyMonitoring_FirePyYieldEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, *PyObject* *retval)

Fire a `PY_YIELD` event.

int **PyMonitoring_FireCallEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, *PyObject* *callable, *PyObject* *arg0)

Fire a `CALL` event.

int **PyMonitoring_FireLineEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, int lineno)

Fire a `LINE` event.

int **PyMonitoring_FireJumpEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, *PyObject* *target_offset)

Fire a `JUMP` event.

int **PyMonitoring_FireBranchEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, *PyObject* *target_offset)

Fire a `BRANCH` event.

int **PyMonitoring_FireCReturnEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, *PyObject* *retval)

Fire a `C_RETURN` event.

int **PyMonitoring_FirePyThrowEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset)

Fire a `PY_THROW` event with the current exception (as returned by `PyErr_GetRaisedException()`).

int **PyMonitoring_FireRaiseEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset)

Fire a `RAISE` event with the current exception (as returned by `PyErr_GetRaisedException()`).

int **PyMonitoring_FireCRaiseEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset)

Fire a `C_RAISE` event with the current exception (as returned by `PyErr_GetRaisedException()`).

int **PyMonitoring_FireReraiseEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset)

Fire a `RERAISE` event with the current exception (as returned by `PyErr_GetRaisedException()`).

int **PyMonitoring_FireExceptionHandledEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset)

Fire an `EXCEPTION_HANDLED` event with the current exception (as returned by `PyErr_GetRaisedException()`).

int **PyMonitoring_FirePyUnwindEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset)

Fire a `PY_UNWIND` event with the current exception (as returned by `PyErr_GetRaisedException()`).

int **PyMonitoring_FireStopIterationEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, *PyObject* *value)

Fire a `STOP_ITERATION` event. If `value` is an instance of `StopIteration`, it is used. Otherwise, a new `StopIteration` instance is created with `value` as its argument.

15.1 Managing the Monitoring State

Monitoring states can be managed with the help of monitoring scopes. A scope would typically correspond to a python function.

int **PyMonitoring_EnterScope**(*PyMonitoringState* *state_array, uint64_t *version, const uint8_t *event_types, *Py_ssize_t* length)

Enter a monitored scope. `event_types` is an array of the event IDs for events that may be fired from the scope. For example, the ID of a `PY_START` event is the value `PY_MONITORING_EVENT_PY_START`, which is numerically equal to the base-2 logarithm of `sys.monitoring.events.PY_START`. `state_array` is an array with a monitoring state entry for each event in `event_types`, it is allocated by the user but populated by `PyMonitoring_EnterScope()` with information about the activation state of the event. The size of `event_types` (and hence also of `state_array`) is given in `length`.

The `version` argument is a pointer to a value which should be allocated by the user together with `state_array` and initialized to 0, and then set only by `PyMonitoring_EnterScope()` itelf. It allows this function to determine whether event states have changed since the previous call, and to return quickly if they have not.

The scopes referred to here are lexical scopes: a function, class or method. `PyMonitoring_EnterScope()` should be called whenever the lexical scope is entered. Scopes can be reentered, reusing the same *state_array* and *version*, in situations like when emulating a recursive Python function. When a code-like's execution is paused, such as when emulating a generator, the scope needs to be exited and re-entered.

The macros for *event_types* are:

Macro	Zdarzenie
PY_MONITORING_EVENT_BRANCH	BRANCH
PY_MONITORING_EVENT_CALL	CALL
PY_MONITORING_EVENT_C_RAISE	C_RAISE
PY_MONITORING_EVENT_C_RETURN	C_RETURN
PY_MONITORING_EVENT_EXCEPTION_HANDLED	EXCEPTION_HANDLED
PY_MONITORING_EVENT_INSTRUCTION	INSTRUCTION
PY_MONITORING_EVENT_JUMP	JUMP
PY_MONITORING_EVENT_LINE	LINE
PY_MONITORING_EVENT_PY_RESUME	PY_RESUME
PY_MONITORING_EVENT_PY_RETURN	PY_RETURN
PY_MONITORING_EVENT_PY_START	PY_START
PY_MONITORING_EVENT_PY_THROW	PY_THROW
PY_MONITORING_EVENT_PY_UNWIND	PY_UNWIND
PY_MONITORING_EVENT_PY_YIELD	PY_YIELD
PY_MONITORING_EVENT_RAISE	RAISE
PY_MONITORING_EVENT_RERAISE	RERAISE
PY_MONITORING_EVENT_STOP_ITERATION	STOP_ITERATION

int **PyMonitoring_ExitScope**(void)

Exit the last scope that was entered with `PyMonitoring_EnterScope()`.

DODATEK A

Słownik

>>>

The default Python prompt of the *interactive* shell. Often seen for code examples which can be executed interactively in the interpreter.

...

Może odnosić się do:

- The default Python prompt of the *interactive* shell when entering the code for an indented code block, when within a pair of matching left and right delimiters (parentheses, square brackets, curly braces or triple quotes), or after specifying a decorator.
- Wbudowanej stałej `Ellipsis`.

abstrakcyjna klasa bazowa

Abstract base classes complement *duck-typing* by providing a way to define interfaces when other techniques like `hasattr()` would be clumsy or subtly wrong (for example with magic methods). ABCs introduce virtual subclasses, which are classes that don't inherit from a class but are still recognized by `isinstance()` and `issubclass()`; see the `abc` module documentation. Python comes with many built-in ABCs for data structures (in the `collections.abc` module), numbers (in the `numbers` module), streams (in the `io` module), import finders and loaders (in the `importlib.abc` module). You can create your own ABCs with the `abc` module.

adnotacja

Etykieta powiązana ze zmienną, atrybutem klasy lub parametrem funkcji lub wartością zwracaną, używana zgodnie z konwencją jako *type hint*.

Annotations of local variables cannot be accessed at runtime, but annotations of global variables, class attributes, and functions are stored in the `__annotations__` special attribute of modules, classes, and functions, respectively.

See *variable annotation*, *function annotation*, **PEP 484** and **PEP 526**, which describe this functionality. Also see annotations-howto for best practices on working with annotations.

argument

A value passed to a *function* (or *method*) when calling the function. There are two kinds of argument:

- *keyword argument*: an argument preceded by an identifier (e.g. `name=`) in a function call or passed as a value in a dictionary preceded by `**`. For example, `3` and `5` are both keyword arguments in the following calls to `complex()`:

```
complex(real=3, imag=5)
complex(**{'real': 3, 'imag': 5})
```

- *positional argument*: an argument that is not a keyword argument. Positional arguments can appear at the beginning of an argument list and/or be passed as elements of an *iterable* preceded by `*`. For example, `3` and `5` are both positional arguments in the following calls:

```
complex(3, 5)
complex(*(3, 5))
```

Arguments are assigned to the named local variables in a function body. See the calls section for the rules governing this assignment. Syntactically, any expression can be used to represent an argument; the evaluated value is assigned to the local variable.

See also the *parameter* glossary entry, the FAQ question on the difference between arguments and parameters, and **PEP 362**.

asynchronous context manager
An object which controls the environment seen in an `async with` statement by defining `__aenter__()` and `__aexit__()` methods. Introduced by **PEP 492**.

asynchronous generator
A function which returns an *asynchronous generator iterator*. It looks like a coroutine function defined with `async def` except that it contains `yield` expressions for producing a series of values usable in an `async for` loop.

Usually refers to an asynchronous generator function, but may refer to an *asynchronous generator iterator* in some contexts. In cases where the intended meaning isn't clear, using the full terms avoids ambiguity.

An asynchronous generator function may contain `await` expressions as well as `async for`, and `async with` statements.

asynchronous generator iterator
An object created by a *asynchronous generator* function.

This is an *asynchronous iterator* which when called using the `__anext__()` method returns an awaitable object which will execute the body of the asynchronous generator function until the next `yield` expression.

Each `yield` temporarily suspends processing, remembering the location execution state (including local variables and pending try-statements). When the *asynchronous generator iterator* effectively resumes with another awaitable returned by `__anext__()`, it picks up where it left off. See **PEP 492** and **PEP 525**.

asynchronous iterable
An object, that can be used in an `async for` statement. Must return an *asynchronous iterator* from its `__aiter__()` method. Introduced by **PEP 492**.

asynchronous iterator
An object that implements the `__aiter__()` and `__anext__()` methods. `__anext__()` must return an *awaitable* object. `async for` resolves the awaitables returned by an asynchronous iterator's `__anext__()` method until it raises a `StopAsyncIteration` exception. Introduced by **PEP 492**.

atrybut
A value associated with an object which is usually referenced by name using dotted expressions. For example, if an object *o* has an attribute *a* it would be referenced as *o.a*.

It is possible to give an object an attribute whose name is not an identifier as defined by identifiers, for example using `setattr()`, if the object allows it. Such an attribute will not be accessible using a dotted expression, and would instead need to be retrieved with `getattr()`.

awaitable
An object that can be used in an `await` expression. Can be a *coroutine* or an object with an `__await__()` method. See also **PEP 492**.

BDFL
Benevolent Dictator For Life, a.k.a. Guido van Rossum, Python's creator.

binary file

A *file object* able to read and write *bytes-like objects*. Examples of binary files are files opened in binary mode (`'rb'`, `'wb'` or `'rb+'`), `sys.stdin.buffer`, `sys.stdout.buffer`, and instances of `io.BytesIO` and `gzip.GzipFile`.

See also *text file* for a file object able to read and write `str` objects.

borrowed reference

In Python's C API, a borrowed reference is a reference to an object, where the code using the object does not own the reference. It becomes a dangling pointer if the object is destroyed. For example, a garbage collection can remove the last *strong reference* to the object and so destroy it.

Calling `Py_INCREF()` on the *borrowed reference* is recommended to convert it to a *strong reference* in-place, except when the object cannot be destroyed before the last usage of the borrowed reference. The `Py_NewRef()` function can be used to create a new *strong reference*.

bytes-like object

An object that supports the *Buffer Protocol* and can export a C-*contiguous* buffer. This includes all `bytes`, `bytearray`, and `array.array` objects, as well as many common `memoryview` objects. Bytes-like objects can be used for various operations that work with binary data; these include compression, saving to a binary file, and sending over a socket.

Some operations need the binary data to be mutable. The documentation often refers to these as „read-write bytes-like objects". Example mutable buffer objects include `bytearray` and a `memoryview` of a `bytearray`. Other operations require the binary data to be stored in immutable objects („read-only bytes-like objects"); examples of these include `bytes` and a `memoryview` of a `bytes` object.

kod bajtowy

Python source code is compiled into bytecode, the internal representation of a Python program in the CPython interpreter. The bytecode is also cached in `.pyc` files so that executing the same file is faster the second time (recompilation from source to bytecode can be avoided). This „intermediate language" is said to run on a *virtual machine* that executes the machine code corresponding to each bytecode. Do note that bytecodes are not expected to work between different Python virtual machines, nor to be stable between Python releases.

A list of bytecode instructions can be found in the documentation for the dis module.

callable

A callable is an object that can be called, possibly with a set of arguments (see *argument*), with the following syntax:

```
callable(argument1, argument2, argumentN)
```

A *function*, and by extension a *method*, is a callable. An instance of a class that implements the `__call__()` method is also a callable.

wywołanie zwrotne

A subroutine function which is passed as an argument to be executed at some point in the future.

class

A template for creating user-defined objects. Class definitions normally contain method definitions which operate on instances of the class.

class variable

A variable defined in a class and intended to be modified only at class level (i.e., not in an instance of the class).

closure variable

A *free variable* referenced from a *nested scope* that is defined in an outer scope rather than being resolved at runtime from the globals or builtin namespaces. May be explicitly defined with the `nonlocal` keyword to allow write access, or implicitly defined if the variable is only being read.

For example, in the `inner` function in the following code, both `x` and `print` are *free variables*, but only `x` is a *closure variable*:

```
def outer():
    x = 0
    def inner():
        nonlocal x
        x += 1
        print(x)
    return inner
```

Due to the `codeobject.co_freevars` attribute (which, despite its name, only includes the names of closure variables rather than listing all referenced free variables), the more general *free variable* term is sometimes used even when the intended meaning is to refer specifically to closure variables.

liczba zespolona
An extension of the familiar real number system in which all numbers are expressed as a sum of a real part and an imaginary part. Imaginary numbers are real multiples of the imaginary unit (the square root of `-1`), often written `i` in mathematics or `j` in engineering. Python has built-in support for complex numbers, which are written with this latter notation; the imaginary part is written with a `j` suffix, e.g., `3+1j`. To get access to complex equivalents of the `math` module, use `cmath`. Use of complex numbers is a fairly advanced mathematical feature. If you're not aware of a need for them, it's almost certain you can safely ignore them.

context manager
An object which controls the environment seen in a `with` statement by defining `__enter__()` and `__exit__()` methods. See **PEP 343**.

context variable
A variable which can have different values depending on its context. This is similar to Thread-Local Storage in which each execution thread may have a different value for a variable. However, with context variables, there may be several contexts in one execution thread and the main usage for context variables is to keep track of variables in concurrent asynchronous tasks. See `contextvars`.

contiguous
A buffer is considered contiguous exactly if it is either *C-contiguous* or *Fortran contiguous*. Zero-dimensional buffers are C and Fortran contiguous. In one-dimensional arrays, the items must be laid out in memory next to each other, in order of increasing indexes starting from zero. In multidimensional C-contiguous arrays, the last index varies the fastest when visiting items in order of memory address. However, in Fortran contiguous arrays, the first index varies the fastest.

coroutine
Coroutines are a more generalized form of subroutines. Subroutines are entered at one point and exited at another point. Coroutines can be entered, exited, and resumed at many different points. They can be implemented with the `async def` statement. See also **PEP 492**.

coroutine function
A function which returns a *coroutine* object. A coroutine function may be defined with the `async def` statement, and may contain `await`, `async for`, and `async with` keywords. These were introduced by **PEP 492**.

CPython
The canonical implementation of the Python programming language, as distributed on python.org. The term „CPython" is used when necessary to distinguish this implementation from others such as Jython or IronPython.

decorator
A function returning another function, usually applied as a function transformation using the `@wrapper` syntax. Common examples for decorators are `classmethod()` and `staticmethod()`.

The decorator syntax is merely syntactic sugar, the following two function definitions are semantically equivalent:

```
def f(arg):
    ...
f = staticmethod(f)
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
@staticmethod
def f(arg):
    ...
```

The same concept exists for classes, but is less commonly used there. See the documentation for function definitions and class definitions for more about decorators.

descriptor
Any object which defines the methods `__get__()`, `__set__()`, or `__delete__()`. When a class attribute is a descriptor, its special binding behavior is triggered upon attribute lookup. Normally, using *a.b* to get, set or delete an attribute looks up the object named *b* in the class dictionary for *a*, but if *b* is a descriptor, the respective descriptor method gets called. Understanding descriptors is a key to a deep understanding of Python because they are the basis for many features including functions, methods, properties, class methods, static methods, and reference to super classes.

For more information about descriptors» methods, see descriptors or the Descriptor How To Guide.

słownik
An associative array, where arbitrary keys are mapped to values. The keys can be any object with `__hash__()` and `__eq__()` methods. Called a hash in Perl.

dictionary comprehension
A compact way to process all or part of the elements in an iterable and return a dictionary with the results. `results = {n: n ** 2 for n in range(10)}` generates a dictionary containing key `n` mapped to value `n ** 2`. See comprehensions.

dictionary view
The objects returned from `dict.keys()`, `dict.values()`, and `dict.items()` are called dictionary views. They provide a dynamic view on the dictionary's entries, which means that when the dictionary changes, the view reflects these changes. To force the dictionary view to become a full list use `list(dictview)`. See dict-views.

docstring
A string literal which appears as the first expression in a class, function or module. While ignored when the suite is executed, it is recognized by the compiler and put into the `__doc__` attribute of the enclosing class, function or module. Since it is available via introspection, it is the canonical place for documentation of the object.

duck-typing
A programming style which does not look at an object's type to determine if it has the right interface; instead, the method or attribute is simply called or used („If it looks like a duck and quacks like a duck, it must be a duck.") By emphasizing interfaces rather than specific types, well-designed code improves its flexibility by allowing polymorphic substitution. Duck-typing avoids tests using `type()` or `isinstance()`. (Note, however, that duck-typing can be complemented with *abstract base classes*.) Instead, it typically employs `hasattr()` tests or *EAFP* programming.

EAFP
Easier to ask for forgiveness than permission. This common Python coding style assumes the existence of valid keys or attributes and catches exceptions if the assumption proves false. This clean and fast style is characterized by the presence of many `try` and `except` statements. The technique contrasts with the *LBYL* style common to many other languages such as C.

expression
A piece of syntax which can be evaluated to some value. In other words, an expression is an accumulation of expression elements like literals, names, attribute access, operators or function calls which all return a value. In contrast to many other languages, not all language constructs are expressions. There are also *statement*s which cannot be used as expressions, such as `while`. Assignments are also statements, not expressions.

moduł rozszerzenia
A module written in C or C++, using Python's C API to interact with the core and with user code.

f-string
String literals prefixed with `'f'` or `'F'` are commonly called „f-strings" which is short for formatted string literals. See also **PEP 498**.

file object
An object exposing a file-oriented API (with methods such as `read()` or `write()`) to an underlying resource. Depending on the way it was created, a file object can mediate access to a real on-disk file or to another type of storage or communication device (for example standard input/output, in-memory buffers, sockets, pipes, etc.). File objects are also called *file-like objects* or *streams*.

There are actually three categories of file objects: raw *binary files*, buffered *binary files* and *text files*. Their interfaces are defined in the `io` module. The canonical way to create a file object is by using the `open()` function.

file-like object
A synonym for *file object*.

filesystem encoding and error handler
Encoding and error handler used by Python to decode bytes from the operating system and encode Unicode to the operating system.

The filesystem encoding must guarantee to successfully decode all bytes below 128. If the file system encoding fails to provide this guarantee, API functions can raise `UnicodeError`.

The `sys.getfilesystemencoding()` and `sys.getfilesystemencodeerrors()` functions can be used to get the filesystem encoding and error handler.

The *filesystem encoding and error handler* are configured at Python startup by the `PyConfig_Read()` function: see `filesystem_encoding` and `filesystem_errors` members of `PyConfig`.

See also the *locale encoding*.

finder
An object that tries to find the *loader* for a module that is being imported.

There are two types of finder: *meta path finders* for use with `sys.meta_path`, and *path entry finders* for use with `sys.path_hooks`.

See finders-and-loaders and `importlib` for much more detail.

floor division
Mathematical division that rounds down to nearest integer. The floor division operator is `//`. For example, the expression `11 // 4` evaluates to `2` in contrast to the `2.75` returned by float true division. Note that `(-11) // 4` is `-3` because that is `-2.75` rounded *downward*. See **PEP 238**.

free threading
A threading model where multiple threads can run Python bytecode simultaneously within the same interpreter. This is in contrast to the *global interpreter lock* which allows only one thread to execute Python bytecode at a time. See **PEP 703**.

free variable
Formally, as defined in the language execution model, a free variable is any variable used in a namespace which is not a local variable in that namespace. See *closure variable* for an example. Pragmatically, due to the name of the `codeobject.co_freevars` attribute, the term is also sometimes used as a synonym for *closure variable*.

funkcja
A series of statements which returns some value to a caller. It can also be passed zero or more *arguments* which may be used in the execution of the body. See also *parameter*, *method*, and the function section.

function annotation
An *annotation* of a function parameter or return value.

Function annotations are usually used for *type hints*: for example, this function is expected to take two `int` arguments and is also expected to have an `int` return value:

```
def sum_two_numbers(a: int, b: int) -> int:
   return a + b
```

Function annotation syntax is explained in section function.

See *variable annotation* and **PEP 484**, which describe this functionality. Also see annotations-howto for best practices on working with annotations.

__future__

A future statement, `from __future__ import <feature>`, directs the compiler to compile the current module using syntax or semantics that will become standard in a future release of Python. The `__future__` module documents the possible values of *feature*. By importing this module and evaluating its variables, you can see when a new feature was first added to the language and when it will (or did) become the default:

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

garbage collection

The process of freeing memory when it is not used anymore. Python performs garbage collection via reference counting and a cyclic garbage collector that is able to detect and break reference cycles. The garbage collector can be controlled using the `gc` module.

generator

A function which returns a *generator iterator*. It looks like a normal function except that it contains `yield` expressions for producing a series of values usable in a for-loop or that can be retrieved one at a time with the `next()` function.

Usually refers to a generator function, but may refer to a *generator iterator* in some contexts. In cases where the intended meaning isn't clear, using the full terms avoids ambiguity.

generator iterator

An object created by a *generator* function.

Each `yield` temporarily suspends processing, remembering the location execution state (including local variables and pending try-statements). When the *generator iterator* resumes, it picks up where it left off (in contrast to functions which start fresh on every invocation).

generator expression

An *expression* that returns an *iterator*. It looks like a normal expression followed by a `for` clause defining a loop variable, range, and an optional `if` clause. The combined expression generates values for an enclosing function:

```
>>> sum(i*i for i in range(10))         # sum of squares 0, 1, 4, ... 81
285
```

generic function

A function composed of multiple functions implementing the same operation for different types. Which implementation should be used during a call is determined by the dispatch algorithm.

See also the *single dispatch* glossary entry, the `functools.singledispatch()` decorator, and **PEP 443**.

generic type

A *type* that can be parameterized; typically a container class such as `list` or `dict`. Used for *type hints* and *annotations*.

For more details, see generic alias types, **PEP 483**, **PEP 484**, **PEP 585**, and the `typing` module.

GIL

See *global interpreter lock*.

global interpreter lock

The mechanism used by the *CPython* interpreter to assure that only one thread executes Python *bytecode* at a time. This simplifies the CPython implementation by making the object model (including critical built-in types such as `dict`) implicitly safe against concurrent access. Locking the entire interpreter makes it easier

for the interpreter to be multi-threaded, at the expense of much of the parallelism afforded by multi-processor machines.

However, some extension modules, either standard or third-party, are designed so as to release the GIL when doing computationally intensive tasks such as compression or hashing. Also, the GIL is always released when doing I/O.

As of Python 3.13, the GIL can be disabled using the `--disable-gil` build configuration. After building Python with this option, code must be run with `-X gil=0` or after setting the `PYTHON_GIL=0` environment variable. This feature enables improved performance for multi-threaded applications and makes it easier to use multi-core CPUs efficiently. For more details, see **PEP 703**.

hash-based pyc
A bytecode cache file that uses the hash rather than the last-modified time of the corresponding source file to determine its validity. See pyc-invalidation.

hashable
An object is *hashable* if it has a hash value which never changes during its lifetime (it needs a `__hash__()` method), and can be compared to other objects (it needs an `__eq__()` method). Hashable objects which compare equal must have the same hash value.

Hashability makes an object usable as a dictionary key and a set member, because these data structures use the hash value internally.

Most of Python's immutable built-in objects are hashable; mutable containers (such as lists or dictionaries) are not; immutable containers (such as tuples and frozensets) are only hashable if their elements are hashable. Objects which are instances of user-defined classes are hashable by default. They all compare unequal (except with themselves), and their hash value is derived from their `id()`.

IDLE
An Integrated Development and Learning Environment for Python. idle is a basic editor and interpreter environment which ships with the standard distribution of Python.

immortal
Immortal objects are a CPython implementation detail introduced in **PEP 683**.

If an object is immortal, its *reference count* is never modified, and therefore it is never deallocated while the interpreter is running. For example, `True` and `None` are immortal in CPython.

immutable
An object with a fixed value. Immutable objects include numbers, strings and tuples. Such an object cannot be altered. A new object has to be created if a different value has to be stored. They play an important role in places where a constant hash value is needed, for example as a key in a dictionary.

import path
A list of locations (or *path entries*) that are searched by the *path based finder* for modules to import. During import, this list of locations usually comes from `sys.path`, but for subpackages it may also come from the parent package's `__path__` attribute.

importing
The process by which Python code in one module is made available to Python code in another module.

importer
An object that both finds and loads a module; both a *finder* and *loader* object.

interactive
Python has an interactive interpreter which means you can enter statements and expressions at the interpreter prompt, immediately execute them and see their results. Just launch `python` with no arguments (possibly by selecting it from your computer's main menu). It is a very powerful way to test out new ideas or inspect modules and packages (remember `help(x)`). For more on interactive mode, see tut-interac.

interpreted
Python is an interpreted language, as opposed to a compiled one, though the distinction can be blurry because of the presence of the bytecode compiler. This means that source files can be run directly without explicitly creating an executable which is then run. Interpreted languages typically have a shorter development/debug cycle than compiled ones, though their programs generally also run more slowly. See also *interactive*.

interpreter shutdown

When asked to shut down, the Python interpreter enters a special phase where it gradually releases all allocated resources, such as modules and various critical internal structures. It also makes several calls to the *garbage collector*. This can trigger the execution of code in user-defined destructors or weakref callbacks. Code executed during the shutdown phase can encounter various exceptions as the resources it relies on may not function anymore (common examples are library modules or the warnings machinery).

The main reason for interpreter shutdown is that the `__main__` module or the script being run has finished executing.

iterable

An object capable of returning its members one at a time. Examples of iterables include all sequence types (such as `list`, `str`, and `tuple`) and some non-sequence types like `dict`, *file objects*, and objects of any classes you define with an `__iter__()` method or with a `__getitem__()` method that implements *sequence* semantics.

Iterables can be used in a `for` loop and in many other places where a sequence is needed (`zip()`, `map()`, ...). When an iterable object is passed as an argument to the built-in function `iter()`, it returns an iterator for the object. This iterator is good for one pass over the set of values. When using iterables, it is usually not necessary to call `iter()` or deal with iterator objects yourself. The `for` statement does that automatically for you, creating a temporary unnamed variable to hold the iterator for the duration of the loop. See also *iterator*, *sequence*, and *generator*.

iterator

An object representing a stream of data. Repeated calls to the iterator's `__next__()` method (or passing it to the built-in function `next()`) return successive items in the stream. When no more data are available a `StopIteration` exception is raised instead. At this point, the iterator object is exhausted and any further calls to its `__next__()` method just raise `StopIteration` again. Iterators are required to have an `__iter__()` method that returns the iterator object itself so every iterator is also iterable and may be used in most places where other iterables are accepted. One notable exception is code which attempts multiple iteration passes. A container object (such as a `list`) produces a fresh new iterator each time you pass it to the `iter()` function or use it in a `for` loop. Attempting this with an iterator will just return the same exhausted iterator object used in the previous iteration pass, making it appear like an empty container.

More information can be found in typeiter.

Szczegół implementacyjny CPythona: CPython does not consistently apply the requirement that an iterator define `__iter__()`. And also please note that the free-threading CPython does not guarantee the thread-safety of iterator operations.

key function

A key function or collation function is a callable that returns a value used for sorting or ordering. For example, `locale.strxfrm()` is used to produce a sort key that is aware of locale specific sort conventions.

A number of tools in Python accept key functions to control how elements are ordered or grouped. They include `min()`, `max()`, `sorted()`, `list.sort()`, `heapq.merge()`, `heapq.nsmallest()`, `heapq.nlargest()`, and `itertools.groupby()`.

There are several ways to create a key function. For example. the `str.lower()` method can serve as a key function for case insensitive sorts. Alternatively, a key function can be built from a `lambda` expression such as `lambda r: (r[0], r[2])`. Also, `operator.attrgetter()`, `operator.itemgetter()`, and `operator.methodcaller()` are three key function constructors. See the Sorting HOW TO for examples of how to create and use key functions.

keyword argument

See *argument*.

lambda

An anonymous inline function consisting of a single *expression* which is evaluated when the function is called. The syntax to create a lambda function is `lambda [parameters]: expression`

LBYL

Look before you leap. This coding style explicitly tests for pre-conditions before making calls or lookups. This style contrasts with the *EAFP* approach and is characterized by the presence of many `if` statements.

In a multi-threaded environment, the LBYL approach can risk introducing a race condition between „the looking" and „the leaping". For example, the code, `if key in mapping: return mapping[key]` can fail if another thread removes *key* from *mapping* after the test, but before the lookup. This issue can be solved with locks or by using the EAFP approach.

list
A built-in Python *sequence*. Despite its name it is more akin to an array in other languages than to a linked list since access to elements is $O(1)$.

list comprehension
A compact way to process all or part of the elements in a sequence and return a list with the results. `result = ['{:#04x}'.format(x) for x in range(256) if x % 2 == 0]` generates a list of strings containing even hex numbers (0x..) in the range from 0 to 255. The `if` clause is optional. If omitted, all elements in `range(256)` are processed.

loader
An object that loads a module. It must define a method named `load_module()`. A loader is typically returned by a *finder*. See also:

- finders-and-loaders
- `importlib.abc.Loader`
- **PEP 302**

locale encoding
On Unix, it is the encoding of the LC_CTYPE locale. It can be set with `locale.setlocale(locale.LC_CTYPE, new_locale)`.

On Windows, it is the ANSI code page (ex: `"cp1252"`).

On Android and VxWorks, Python uses `"utf-8"` as the locale encoding.

`locale.getencoding()` can be used to get the locale encoding.

See also the *filesystem encoding and error handler*.

magic method
An informal synonym for *special method*.

mapping
A container object that supports arbitrary key lookups and implements the methods specified in the `collections.abc.Mapping` or `collections.abc.MutableMapping` abstract base classes. Examples include `dict`, `collections.defaultdict`, `collections.OrderedDict` and `collections.Counter`.

meta path finder
A *finder* returned by a search of `sys.meta_path`. Meta path finders are related to, but different from *path entry finders*.

See `importlib.abc.MetaPathFinder` for the methods that meta path finders implement.

metaclass
The class of a class. Class definitions create a class name, a class dictionary, and a list of base classes. The metaclass is responsible for taking those three arguments and creating the class. Most object oriented programming languages provide a default implementation. What makes Python special is that it is possible to create custom metaclasses. Most users never need this tool, but when the need arises, metaclasses can provide powerful, elegant solutions. They have been used for logging attribute access, adding thread-safety, tracking object creation, implementing singletons, and many other tasks.

More information can be found in metaclasses.

method
A function which is defined inside a class body. If called as an attribute of an instance of that class, the method will get the instance object as its first *argument* (which is usually called `self`). See *function* and *nested scope*.

method resolution order
Method Resolution Order is the order in which base classes are searched for a member during lookup. See python_2.3_mro for details of the algorithm used by the Python interpreter since the 2.3 release.

moduł
An object that serves as an organizational unit of Python code. Modules have a namespace containing arbitrary Python objects. Modules are loaded into Python by the process of *importing*.

See also *package*.

module spec
A namespace containing the import-related information used to load a module. An instance of `importlib.machinery.ModuleSpec`.

See also module-specs.

MRO
See *method resolution order*.

mutable
Mutable objects can change their value but keep their `id()`. See also *immutable*.

named tuple
The term „named tuple" applies to any type or class that inherits from tuple and whose indexable elements are also accessible using named attributes. The type or class may have other features as well.

Several built-in types are named tuples, including the values returned by `time.localtime()` and `os.stat()`. Another example is `sys.float_info`:

```
>>> sys.float_info[1]                    # indexed access
1024
>>> sys.float_info.max_exp               # named field access
1024
>>> isinstance(sys.float_info, tuple)    # kind of tuple
True
```

Some named tuples are built-in types (such as the above examples). Alternatively, a named tuple can be created from a regular class definition that inherits from `tuple` and that defines named fields. Such a class can be written by hand, or it can be created by inheriting `typing.NamedTuple`, or with the factory function `collections.namedtuple()`. The latter techniques also add some extra methods that may not be found in hand-written or built-in named tuples.

namespace
The place where a variable is stored. Namespaces are implemented as dictionaries. There are the local, global and built-in namespaces as well as nested namespaces in objects (in methods). Namespaces support modularity by preventing naming conflicts. For instance, the functions `builtins.open` and `os.open()` are distinguished by their namespaces. Namespaces also aid readability and maintainability by making it clear which module implements a function. For instance, writing `random.seed()` or `itertools.islice()` makes it clear that those functions are implemented by the `random` and `itertools` modules, respectively.

namespace package
A **PEP 420** *package* which serves only as a container for subpackages. Namespace packages may have no physical representation, and specifically are not like a *regular package* because they have no `__init__.py` file.

See also *module*.

nested scope
The ability to refer to a variable in an enclosing definition. For instance, a function defined inside another function can refer to variables in the outer function. Note that nested scopes by default work only for reference and not for assignment. Local variables both read and write in the innermost scope. Likewise, global variables read and write to the global namespace. The `nonlocal` allows writing to outer scopes.

new-style class
Old name for the flavor of classes now used for all class objects. In earlier Python versions, only new-style classes

could use Python's newer, versatile features like `__slots__`, descriptors, properties, `__getattribute__()`, class methods, and static methods.

obiekt
Any data with state (attributes or value) and defined behavior (methods). Also the ultimate base class of any *new-style class*.

optimized scope
A scope where target local variable names are reliably known to the compiler when the code is compiled, allowing optimization of read and write access to these names. The local namespaces for functions, generators, coroutines, comprehensions, and generator expressions are optimized in this fashion. Note: most interpreter optimizations are applied to all scopes, only those relying on a known set of local and nonlocal variable names are restricted to optimized scopes.

package
A Python *module* which can contain submodules or recursively, subpackages. Technically, a package is a Python module with a `__path__` attribute.

See also *regular package* and *namespace package*.

parameter
A named entity in a *function* (or method) definition that specifies an *argument* (or in some cases, arguments) that the function can accept. There are five kinds of parameter:

- *positional-or-keyword*: specifies an argument that can be passed either *positionally* or as a *keyword argument*. This is the default kind of parameter, for example *foo* and *bar* in the following:

```
def func(foo, bar=None): ...
```

- *positional-only*: specifies an argument that can be supplied only by position. Positional-only parameters can be defined by including a `/` character in the parameter list of the function definition after them, for example *posonly1* and *posonly2* in the following:

```
def func(posonly1, posonly2, /, positional_or_keyword): ...
```

- *keyword-only*: specifies an argument that can be supplied only by keyword. Keyword-only parameters can be defined by including a single var-positional parameter or bare `*` in the parameter list of the function definition before them, for example *kw_only1* and *kw_only2* in the following:

```
def func(arg, *, kw_only1, kw_only2): ...
```

- *var-positional*: specifies that an arbitrary sequence of positional arguments can be provided (in addition to any positional arguments already accepted by other parameters). Such a parameter can be defined by prepending the parameter name with `*`, for example *args* in the following:

```
def func(*args, **kwargs): ...
```

- *var-keyword*: specifies that arbitrarily many keyword arguments can be provided (in addition to any keyword arguments already accepted by other parameters). Such a parameter can be defined by prepending the parameter name with `**`, for example *kwargs* in the example above.

Parameters can specify both optional and required arguments, as well as default values for some optional arguments.

See also the *argument* glossary entry, the FAQ question on the difference between arguments and parameters, the `inspect.Parameter` class, the function section, and **PEP 362**.

path entry
A single location on the *import path* which the *path based finder* consults to find modules for importing.

path entry finder
A *finder* returned by a callable on `sys.path_hooks` (i.e. a *path entry hook*) which knows how to locate modules given a *path entry*.

See `importlib.abc.PathEntryFinder` for the methods that path entry finders implement.

path entry hook
A callable on the `sys.path_hooks` list which returns a *path entry finder* if it knows how to find modules on a specific *path entry*.

path based finder
One of the default *meta path finders* which searches an *import path* for modules.

path-like object
An object representing a file system path. A path-like object is either a `str` or `bytes` object representing a path, or an object implementing the `os.PathLike` protocol. An object that supports the `os.PathLike` protocol can be converted to a `str` or `bytes` file system path by calling the `os.fspath()` function; `os.fsdecode()` and `os.fsencode()` can be used to guarantee a `str` or `bytes` result instead, respectively. Introduced by **PEP 519**.

PEP
Python Enhancement Proposal. A PEP is a design document providing information to the Python community, or describing a new feature for Python or its processes or environment. PEPs should provide a concise technical specification and a rationale for proposed features.

PEPs are intended to be the primary mechanisms for proposing major new features, for collecting community input on an issue, and for documenting the design decisions that have gone into Python. The PEP author is responsible for building consensus within the community and documenting dissenting opinions.

See **PEP 1**.

portion
A set of files in a single directory (possibly stored in a zip file) that contribute to a namespace package, as defined in **PEP 420**.

positional argument
See *argument*.

provisional API
A provisional API is one which has been deliberately excluded from the standard library's backwards compatibility guarantees. While major changes to such interfaces are not expected, as long as they are marked provisional, backwards incompatible changes (up to and including removal of the interface) may occur if deemed necessary by core developers. Such changes will not be made gratuitously – they will occur only if serious fundamental flaws are uncovered that were missed prior to the inclusion of the API.

Even for provisional APIs, backwards incompatible changes are seen as a „solution of last resort" - every attempt will still be made to find a backwards compatible resolution to any identified problems.

This process allows the standard library to continue to evolve over time, without locking in problematic design errors for extended periods of time. See **PEP 411** for more details.

provisional package
See *provisional API*.

Python 3000
Nickname for the Python 3.x release line (coined long ago when the release of version 3 was something in the distant future.) This is also abbreviated „Py3k".

Pythonic
An idea or piece of code which closely follows the most common idioms of the Python language, rather than implementing code using concepts common to other languages. For example, a common idiom in Python is to loop over all elements of an iterable using a `for` statement. Many other languages don't have this type of construct, so people unfamiliar with Python sometimes use a numerical counter instead:

```
for i in range(len(food)):
    print(food[i])
```

As opposed to the cleaner, Pythonic method:

```
for piece in food:
    print(piece)
```

qualified name

A dotted name showing the „path" from a module's global scope to a class, function or method defined in that module, as defined in **PEP 3155**. For top-level functions and classes, the qualified name is the same as the object's name:

```
>>> class C:
...     class D:
...         def meth(self):
...             pass
...
>>> C.__qualname__
'C'
>>> C.D.__qualname__
'C.D'
>>> C.D.meth.__qualname__
'C.D.meth'
```

When used to refer to modules, the *fully qualified name* means the entire dotted path to the module, including any parent packages, e.g. `email.mime.text`:

```
>>> import email.mime.text
>>> email.mime.text.__name__
'email.mime.text'
```

reference count

The number of references to an object. When the reference count of an object drops to zero, it is deallocated. Some objects are *immortal* and have reference counts that are never modified, and therefore the objects are never deallocated. Reference counting is generally not visible to Python code, but it is a key element of the *CPython* implementation. Programmers can call the `sys.getrefcount()` function to return the reference count for a particular object.

regular package

A traditional *package*, such as a directory containing an `__init__.py` file.

See also *namespace package*.

REPL

An acronym for the „read–eval–print loop", another name for the *interactive* interpreter shell.

__slots__

A declaration inside a class that saves memory by pre-declaring space for instance attributes and eliminating instance dictionaries. Though popular, the technique is somewhat tricky to get right and is best reserved for rare cases where there are large numbers of instances in a memory-critical application.

sequence

An *iterable* which supports efficient element access using integer indices via the `__getitem__()` special method and defines a `__len__()` method that returns the length of the sequence. Some built-in sequence types are `list`, `str`, `tuple`, and `bytes`. Note that `dict` also supports `__getitem__()` and `__len__()`, but is considered a mapping rather than a sequence because the lookups use arbitrary *hashable* keys rather than integers.

The `collections.abc.Sequence` abstract base class defines a much richer interface that goes beyond just `__getitem__()` and `__len__()`, adding `count()`, `index()`, `__contains__()`, and `__reversed__()`. Types that implement this expanded interface can be registered explicitly using `register()`. For more documentation on sequence methods generally, see Common Sequence Operations.

set comprehension

A compact way to process all or part of the elements in an iterable and return a set with the results. `results = {c for c in 'abracadabra' if c not in 'abc'}` generates the set of strings `{'r', 'd'}`. See comprehensions.

single dispatch

A form of *generic function* dispatch where the implementation is chosen based on the type of a single argument.

slice
An object usually containing a portion of a *sequence*. A slice is created using the subscript notation, `[]` with colons between numbers when several are given, such as in `variable_name[1:3:5]`. The bracket (subscript) notation uses `slice` objects internally.

soft deprecated
A soft deprecated API should not be used in new code, but it is safe for already existing code to use it. The API remains documented and tested, but will not be enhanced further.

Soft deprecation, unlike normal deprecation, does not plan on removing the API and will not emit warnings.

See PEP 387: Soft Deprecation.

special method
A method that is called implicitly by Python to execute a certain operation on a type, such as addition. Such methods have names starting and ending with double underscores. Special methods are documented in specialnames.

instrukcja
A statement is part of a suite (a „block" of code). A statement is either an *expression* or one of several constructs with a keyword, such as `if`, `while` or `for`.

static type checker
An external tool that reads Python code and analyzes it, looking for issues such as incorrect types. See also *type hints* and the `typing` module.

strong reference
In Python's C API, a strong reference is a reference to an object which is owned by the code holding the reference. The strong reference is taken by calling `Py_INCREF()` when the reference is created and released with `Py_DECREF()` when the reference is deleted.

The `Py_NewRef()` function can be used to create a strong reference to an object. Usually, the `Py_DECREF()` function must be called on the strong reference before exiting the scope of the strong reference, to avoid leaking one reference.

See also *borrowed reference*.

text encoding
A string in Python is a sequence of Unicode code points (in range `U+0000–U+10FFFF`). To store or transfer a string, it needs to be serialized as a sequence of bytes.

Serializing a string into a sequence of bytes is known as „encoding", and recreating the string from the sequence of bytes is known as „decoding".

There are a variety of different text serialization codecs, which are collectively referred to as „text encodings".

text file
A *file object* able to read and write `str` objects. Often, a text file actually accesses a byte-oriented datastream and handles the *text encoding* automatically. Examples of text files are files opened in text mode (`'r'` or `'w'`), `sys.stdin`, `sys.stdout`, and instances of `io.StringIO`.

See also *binary file* for a file object able to read and write *bytes-like objects*.

triple-quoted string
A string which is bound by three instances of either a quotation mark (") or an apostrophe («). While they don't provide any functionality not available with single-quoted strings, they are useful for a number of reasons. They allow you to include unescaped single and double quotes within a string and they can span multiple lines without the use of the continuation character, making them especially useful when writing docstrings.

type
The type of a Python object determines what kind of object it is; every object has a type. An object's type is accessible as its `__class__` attribute or can be retrieved with `type(obj)`.

type alias
A synonym for a type, created by assigning the type to an identifier.

Type aliases are useful for simplifying *type hints*. For example:

```
def remove_gray_shades(
        colors: list[tuple[int, int, int]]) -> list[tuple[int, int, int]]:
    pass
```

could be made more readable like this:

```
Color = tuple[int, int, int]

def remove_gray_shades(colors: list[Color]) -> list[Color]:
    pass
```

See `typing` and **PEP 484**, which describe this functionality.

type hint

An *annotation* that specifies the expected type for a variable, a class attribute, or a function parameter or return value.

Type hints are optional and are not enforced by Python but they are useful to *static type checkers*. They can also aid IDEs with code completion and refactoring.

Type hints of global variables, class attributes, and functions, but not local variables, can be accessed using `typing.get_type_hints()`.

See `typing` and **PEP 484**, which describe this functionality.

universal newlines

A manner of interpreting text streams in which all of the following are recognized as ending a line: the Unix end-of-line convention `'\n'`, the Windows convention `'\r\n'`, and the old Macintosh convention `'\r'`. See **PEP 278** and **PEP 3116**, as well as `bytes.splitlines()` for an additional use.

variable annotation

An *annotation* of a variable or a class attribute.

When annotating a variable or a class attribute, assignment is optional:

```
class C:
    field: 'annotation'
```

Variable annotations are usually used for *type hints*: for example this variable is expected to take `int` values:

```
count: int = 0
```

Variable annotation syntax is explained in section annassign.

See *function annotation*, **PEP 484** and **PEP 526**, which describe this functionality. Also see annotations-howto for best practices on working with annotations.

virtual environment

A cooperatively isolated runtime environment that allows Python users and applications to install and upgrade Python distribution packages without interfering with the behaviour of other Python applications running on the same system.

See also `venv`.

virtual machine

A computer defined entirely in software. Python's virtual machine executes the *bytecode* emitted by the bytecode compiler.

Zen of Python

Listing of Python design principles and philosophies that are helpful in understanding and using the language. The listing can be found by typing „`import this`" at the interactive prompt.

DODATEK B

O tej dokumentacji

Dokumenty są wygenerowane ze źródeł reStructuredText przez Sphinksa, procesor dokumentów napisany specjalnie dla dokumentacji Pythona.

Rozwój dokumentacji i jej oprzyrządowania jest w całości wysiłkiem wolontariackim, tak samo jak sam Python. Jeśli chcesz wnieść swój wkład, na stronie reporting-bugs znajdziesz informacje jak to zrobić. Nowi wolontariusze są zawsze mile widziani!

Ogromne podziękowania dla:

- Freda L. Drake'a, Jr., twórcy oryginalnego zestawu narzędzi dokumentacji Pythona i autora dużej części jej treści;
- projektu Docutils za stworzenie reStructuredText i pakietu Docutils;
- Fredrika Lundha za jego projekt Alternative Python Reference, z którego Sphinx wziął wiele dobrych pomysłów.

B.1 Współtwórcy dokumentacji Pythona

Wielu ludzi rozwija język Python, bibliotekę standardową Pythona i dokumentację. W Misc/ACKS w źródłach Pythona znajdziesz częściową listę kontrybutorów.

Tylko dzięki wkładowi społeczności Python ma tak wspaniałą dokumentację – dziękujemy!

DODATEK C

Historia i zapisy prawne

C.1 Historia programu

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI, see https://www.cwi.nl/) in the Netherlands as a successor of a language called ABC. Guido remains Python's principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI, see https://www.cnri.reston.va.us/) in Reston, Virginia where he released several versions of the software.

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs team moved to Digital Creations (now Zope Corporation; see https://www.zope.org/). In 2001, the Python Software Foundation (PSF, see https://www.python.org/psf/) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation is a sponsoring member of the PSF.

All Python releases are Open Source (see https://opensource.org/ for the Open Source Definition). Historically, most, but not all, Python releases have also been GPL-compatible; the table below summarizes the various releases.

wydanie	Pochodne po	Rok	Właściciel	Zgodne z Uprawnieniami Ogólnie Powszechnymi (GPL)?
od 0.9.0 do 1.2	nie podano	od 1991 do 1995	CWI	tak
od 1.3 do 1.5.2	1.2	od 1995 do 1999	CNRI	tak
1.6	1.5.2	2000	CNRI	nie
2.0	1.6	2000	BeOpen.com	nie
1.6.1	1.6	2001	CNRI	nie
2.1	2.0 i 1.6.1	2001	Fundacja Programu języka Pytonowskiego (PSF)	nie
2.0.1	2.0 i 1.6.1	2001	Fundacja Programu języka Pytonowskiego (PSF)	tak
2.1.1	2.1 i 2.0.1	2001	Fundacja Programu języka Pytonowskiego (PSF)	tak
2.1.2	2.1.1	2002	Fundacja Programu języka Pytonowskiego (PSF)	tak
2.1.3	2.1.2	2002	Fundacja Programu języka Pytonowskiego (PSF)	tak
2.2 and above	2.1.1	2001-now	Fundacja Programu języka Pytonowskiego (PSF)	tak

Informacja

Zgodność z uprawnieniami ogólnie powszechnymi (w skrócie - z ang. - GPL) nie oznacza, że rozprowadzamy język pytonowski z uprawnieniami ogólnie powszechnymi (w skrócie - z ang. - GPL). Wszystkie uprawnienia dostarczane z językiem pytonowskim, w przeciwieństwie do uprawnień ogólnie powszechnych (w skrócie - z ang. - GPL), pozwalają na rozpowszechnianie programów języka pytonowskiego z wprowadzonymi zmianami bez ustanawiania tych zmian w ramach otwartych źródeł. Uprawnienia zgodne z ogólnie powszechnymi uprawnieniami (w skrócie - z ang. - GPL) pozwalają na łączenie wydań programu języka pytonowskiego z innymi programami które są wydane z uprawnieniami ogólnie powszechnymi (w skrócie - z ang. - GPL). Inne uprawnienia, niezgodne z ogólnie powszechnymi, na to nie zezwalają.

Podziękowania dla wielu ochotników przychodzących z zewnątrz, którzy pracowali pod kierunkiem Gwidona aby umożliwić te wydania programu języka pytonowskiego.

C.2 Zasady i warunki postępowania z Pythonem i ogólnie jego użycia

Python software and documentation are licensed under the *PSF License Agreement*.

Starting with Python 3.8.6, examples, recipes, and other code in the documentation are dual licensed under the PSF License Agreement and the *Zero-Clause BSD license*.

Some software incorporated into Python is under different licenses. The licenses are listed with code falling under that license. See *Licenses and Acknowledgements for Incorporated Software* for an incomplete list of these licenses.

C.2.1 PSF LICENSE AGREEMENT FOR PYTHON 3.13.0

```
1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and
   the Individual or Organization ("Licensee") accessing and otherwise using Python
   3.13.0 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, PSF hereby
```

```
   grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python 3.13.0 alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's notice of
   copyright, i.e., "Copyright © 2001-2024 Python Software Foundation; All Rights
   Reserved" are retained in Python 3.13.0 alone or in any derivative version
   prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 3.13.0 or any part thereof, and wants to make the
   derivative work available to others as provided herein, then Licensee hereby
   agrees to include in any such work a brief summary of the changes made to Python
   3.13.0.

4. PSF is making Python 3.13.0 available to Licensee on an "AS IS" basis.
   PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE
   USE OF PYTHON 3.13.0 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.13.0
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.13.0, OR ANY DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any relationship
   of agency, partnership, or joint venture between PSF and Licensee.  This License
   Agreement does not grant permission to use PSF trademarks or trade name in a
   trademark sense to endorse or promote products or services of Licensee, or any
   third party.

8. By copying, installing or otherwise using Python 3.13.0, Licensee agrees
   to be bound by the terms and conditions of this License Agreement.
```

C.2.2 BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0

BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

```
1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at
   160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization
   ("Licensee") accessing and otherwise using this software in source or binary
   form and its associated documentation ("the Software").

2. Subject to the terms and conditions of this BeOpen Python License Agreement,
   BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license
   to reproduce, analyze, test, perform and/or display publicly, prepare derivative
   works, distribute, and otherwise use the Software alone or in any derivative
   version, provided, however, that the BeOpen Python License is retained in the
   Software, alone or in any derivative version prepared by Licensee.

3. BeOpen is making the Software available to Licensee on an "AS IS" basis.
   BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE
   USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING,
   MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF
   ADVISED OF THE POSSIBILITY THEREOF.

5. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

6. This License Agreement shall be governed by and interpreted in all respects
   by the law of the State of California, excluding conflict of law provisions.
   Nothing in this License Agreement shall be deemed to create any relationship of
   agency, partnership, or joint venture between BeOpen and Licensee.  This License
   Agreement does not grant permission to use BeOpen trademarks or trade names in a
   trademark sense to endorse or promote products or services of Licensee, or any
   third party.  As an exception, the "BeOpen Python" logos available at
   http://www.pythonlabs.com/logos.html may be used according to the permissions
   granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be
   bound by the terms and conditions of this License Agreement.
```

C.2.3 CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1

```
1. This LICENSE AGREEMENT is between the Corporation for National Research
   Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191
   ("CNRI"), and the Individual or Organization ("Licensee") accessing and
   otherwise using Python 1.6.1 software in source or binary form and its
   associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python 1.6.1 alone or in any derivative version,
   provided, however, that CNRI's License Agreement and CNRI's notice of copyright,
   i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All
   Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version
   prepared by Licensee.  Alternately, in lieu of CNRI's License Agreement,
   Licensee may substitute the following text (omitting the quotes): "Python 1.6.1
   is made available subject to the terms and conditions in CNRI's License
   Agreement.  This Agreement together with Python 1.6.1 may be located on the
   internet using the following unique, persistent identifier (known as a handle):
   1895.22/1013.  This Agreement may also be obtained from a proxy server on the
   internet using the following URL: http://hdl.handle.net/1895.22/1013."

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 1.6.1 or any part thereof, and wants to make the derivative
   work available to others as provided herein, then Licensee hereby agrees to
   include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis.  CNRI
   MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF EXAMPLE,
   BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY
   OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF
   PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

7. This License Agreement shall be governed by the federal intellectual property
   law of the United States, including without limitation the federal copyright
   law, and, to the extent such U.S. federal law does not apply, by the law of the
   Commonwealth of Virginia, excluding Virginia's conflict of law provisions.
   Notwithstanding the foregoing, with regard to derivative works based on Python
   1.6.1 that incorporate non-separable material that was previously distributed
   under the GNU General Public License (GPL), the law of the Commonwealth of
   Virginia shall govern this License Agreement only as to issues arising under or
   with respect to Paragraphs 4, 5, and 7 of this License Agreement.  Nothing in
   this License Agreement shall be deemed to create any relationship of agency,
   partnership, or joint venture between CNRI and Licensee.  This License Agreement
   does not grant permission to use CNRI trademarks or trade name in a trademark
   sense to endorse or promote products or services of Licensee, or any third
   party.

8. By clicking on the "ACCEPT" button where indicated, or by copying, installing
   or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and
   conditions of this License Agreement.
```

C.2.4 CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

```
Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The
Netherlands.  All rights reserved.

Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted, provided that
the above copyright notice appear in all copies and that both that copyright
notice and this permission notice appear in supporting documentation, and that
the name of Stichting Mathematisch Centrum or CWI not be used in advertising or
publicity pertaining to distribution of the software without specific, written
prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO
EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT
OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE,
DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS
SOFTWARE.
```

C.2.5 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON 3.13.0 DOCUMENTATION

```
Permission to use, copy, modify, and/or distribute this software for any
purpose with or without fee is hereby granted.
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH
REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT,
INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM
LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3 Licenses and Acknowledgements for Incorporated Software

This section is an incomplete, but growing list of licenses and acknowledgements for third-party software incorporated in the Python distribution.

C.3.1 Mersenne Twister

The `_random` C extension underlying the `random` module includes code based on a download from http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html. The following are the verbatim comments from the original code:

```
A C-program for MT19937, with initialization improved 2002/1/26.
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using init_genrand(seed)
or init_by_array(init_key, key_length).

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

 1. Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

 2. Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in the
    documentation and/or other materials provided with the distribution.

 3. The names of its contributors may not be used to endorse or promote
    products derived from this software without specific prior written
    permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
Any feedback is very welcome.
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)
```

C.3.2 Sockets

The `socket` module uses the functions, `getaddrinfo()`, and `getnameinfo()`, which are coded in separate source files from the WIDE Project, https://www.wide.ad.jp/.

```
Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors
   may be used to endorse or promote products derived from this software
   without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.3 Asynchronous socket services

The `test.support.asynchat` and `test.support.asyncore` modules contain the following notice:

```
Copyright 1996 by Sam Rushing

                         All Rights Reserved

Permission to use, copy, modify, and distribute this software and
its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of Sam
Rushing not be used in advertising or publicity pertaining to
distribution of the software without specific, written prior
permission.

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT OR
CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

C.3.4 Cookie management

The `http.cookies` module contains the following notice:

```
Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

               All Rights Reserved

Permission to use, copy, modify, and distribute this software
and its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Timothy O'Malley  not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR
ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3.5 Execution tracing

The `trace` module contains the following notice:

```
portions copyright 2001, Autonomous Zones Industries, Inc., all rights...
err...  reserved and offered to the public under the terms of the
Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights reserved.


Permission to use, copy, modify, and distribute this Python software and
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
its associated documentation for any purpose without fee is hereby
granted, provided that the above copyright notice appears in all copies,
and that both that copyright notice and this permission notice appear in
supporting documentation, and that the name of neither Automatrix,
Bioreason or Mojam Media be used in advertising or publicity pertaining to
distribution of the software without specific, written prior permission.
```

C.3.6 UUencode and UUdecode functions

The `uu` codec contains the following notice:

```
Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
                       All Rights Reserved
Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. The C
  version is still 5 times faster, though.
- Arguments more compliant with Python standard
```

C.3.7 XML Remote Procedure Calls

The `xmlrpc.client` module contains the following notice:

```
    The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB
Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its
associated documentation, you agree that you have read, understood,
and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and
its associated documentation for any purpose and without fee is
hereby granted, provided that the above copyright notice appears in
all copies, and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Secret Labs AB or the author not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD
TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANT-
ABILITY AND FITNESS.  IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR
BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE
OF THIS SOFTWARE.
```

C.3.8 test_epoll

The `test.test_epoll` module contains the following notice:

```
Copyright (c) 2001-2006 Twisted Matrix Laboratories.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.9 Select kqueue

The `select` module contains the following notice for the kqueue interface:

```
Copyright (c) 2000 Doug White, 2006 James Knight, 2007 Christian Heimes
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.10 SipHash24

The file `Python/pyhash.c` contains Marek Majkowski» implementation of Dan Bernstein's SipHash24 algorithm. It contains the following note:

```
<MIT License>
Copyright (c) 2013  Marek Majkowski <marek@popcount.org>

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.
</MIT License>

Original location:
   https://github.com/majek/csiphash/

Solution inspired by code from:
   Samuel Neves (supercop/crypto_auth/siphash24/little)
   djb (supercop/crypto_auth/siphash24/little2)
   Jean-Philippe Aumasson (https://131002.net/siphash/siphash24.c)
```

C.3.11 strtod and dtoa

The file `Python/dtoa.c`, which supplies C functions dtoa and strtod for conversion of C doubles to and from strings, is derived from the file of the same name by David M. Gay, currently available from https://web.archive.org/web/20220517033456/http://www.netlib.org/fp/dtoa.c. The original file, as retrieved on March 16, 2009, contains the following copyright and licensing notice:

```
/****************************************************************
 *
 * The author of this software is David M. Gay.
 *
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose without fee is hereby granted, provided that this entire notice
 * is included in all copies of any software which is or includes a copy
 * or modification of this software and in all copies of the supporting
 * documentation for such software.
 *
 * THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
 * WARRANTY.  IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
 * REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
 * OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
 *
 ****************************************************************/
```

C.3.12 OpenSSL

The modules `hashlib`, `posix` and `ssl` use the OpenSSL library for added performance if made available by the operating system. Additionally, the Windows and macOS installers for Python may include a copy of the OpenSSL libraries, so we include a copy of the OpenSSL license here. For the OpenSSL 3.0 release, and later releases derived from that, the Apache License v2 applies:

```
                                 Apache License
                           Version 2.0, January 2004
                        https://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction,
   and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by
   the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all
   other entities that control, are controlled by, or are under common
   control with that entity. For the purposes of this definition,
   "control" means (i) the power, direct or indirect, to cause the
   direction or management of such entity, whether by contract or
   otherwise, or (ii) ownership of fifty percent (50%) or more of the
   outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity
   exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications,
   including but not limited to software source code, documentation
   source, and configuration files.

   "Object" form shall mean any form resulting from mechanical
   transformation or translation of a Source form, including but
   not limited to compiled object code, generated documentation,
   and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or
   Object form, made available under the License, as indicated by a
   copyright notice that is included in or attached to the work
   (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object
   form, that is based on (or derived from) the Work and for which the
   editorial revisions, annotations, elaborations, or other modifications
   represent, as a whole, an original work of authorship. For the purposes
   of this License, Derivative Works shall not include works that remain
   separable from, or merely link (or bind by name) to the interfaces of,
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
   the Work and Derivative Works thereof.

   "Contribution" shall mean any work of authorship, including
   the original version of the Work and any modifications or additions
   to that Work or Derivative Works thereof, that is intentionally
   submitted to Licensor for inclusion in the Work by the copyright owner
   or by an individual or Legal Entity authorized to submit on behalf of
   the copyright owner. For the purposes of this definition, "submitted"
   means any form of electronic, verbal, or written communication sent
   to the Licensor or its representatives, including but not limited to
   communication on electronic mailing lists, source code control systems,
   and issue tracking systems that are managed by, or on behalf of, the
   Licensor for the purpose of discussing and improving the Work, but
   excluding communication that is conspicuously marked or otherwise
   designated in writing by the copyright owner as "Not a Contribution."

   "Contributor" shall mean Licensor and any individual or Legal Entity
   on behalf of whom a Contribution has been received by Licensor and
   subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   copyright license to reproduce, prepare Derivative Works of,
   publicly display, publicly perform, sublicense, and distribute the
   Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   (except as stated in this section) patent license to make, have made,
   use, offer to sell, sell, import, and otherwise transfer the Work,
   where such license applies only to those patent claims licensable
   by such Contributor that are necessarily infringed by their
   Contribution(s) alone or by combination of their Contribution(s)
   with the Work to which such Contribution(s) was submitted. If You
   institute patent litigation against any entity (including a
   cross-claim or counterclaim in a lawsuit) alleging that the Work
   or a Contribution incorporated within the Work constitutes direct
   or contributory patent infringement, then any patent licenses
   granted to You under this License for that Work shall terminate
   as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the
   Work or Derivative Works thereof in any medium, with or without
   modifications, and in Source or Object form, provided that You
   meet the following conditions:

   (a) You must give any other recipients of the Work or
       Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices
       stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works
       that You distribute, all copyright, patent, trademark, and
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
        attribution notices from the Source form of the Work,
        excluding those notices that do not pertain to any part of
        the Derivative Works; and

    (d) If the Work includes a "NOTICE" text file as part of its
        distribution, then any Derivative Works that You distribute must
        include a readable copy of the attribution notices contained
        within such NOTICE file, excluding those notices that do not
        pertain to any part of the Derivative Works, in at least one
        of the following places: within a NOTICE text file distributed
        as part of the Derivative Works; within the Source form or
        documentation, if provided along with the Derivative Works; or,
        within a display generated by the Derivative Works, if and
        wherever such third-party notices normally appear. The contents
        of the NOTICE file are for informational purposes only and
        do not modify the License. You may add Your own attribution
        notices within Derivative Works that You distribute, alongside
        or as an addendum to the NOTICE text from the Work, provided
        that such additional attribution notices cannot be construed
        as modifying the License.

    You may add Your own copyright statement to Your modifications and
    may provide additional or different license terms and conditions
    for use, reproduction, or distribution of Your modifications, or
    for any such Derivative Works as a whole, provided Your use,
    reproduction, and distribution of the Work otherwise complies with
    the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,
   any Contribution intentionally submitted for inclusion in the Work
   by You to the Licensor shall be under the terms and conditions of
   this License, without any additional terms or conditions.
   Notwithstanding the above, nothing herein shall supersede or modify
   the terms of any separate license agreement you may have executed
   with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade
   names, trademarks, service marks, or product names of the Licensor,
   except as required for reasonable and customary use in describing the
   origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or
   agreed to in writing, Licensor provides the Work (and each
   Contributor provides its Contributions) on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
   implied, including, without limitation, any warranties or conditions
   of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
   PARTICULAR PURPOSE. You are solely responsible for determining the
   appropriateness of using or redistributing the Work and assume any
   risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
   whether in tort (including negligence), contract, or otherwise,
   unless required by applicable law (such as deliberate and grossly
   negligent acts) or agreed to in writing, shall any Contributor be
   liable to You for damages, including any direct, indirect, special,
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
   incidental, or consequential damages of any character arising as a
   result of this License or out of the use or inability to use the
   Work (including but not limited to damages for loss of goodwill,
   work stoppage, computer failure or malfunction, or any and all
   other commercial damages or losses), even if such Contributor
   has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing
   the Work or Derivative Works thereof, You may choose to offer,
   and charge a fee for, acceptance of support, warranty, indemnity,
   or other liability obligations and/or rights consistent with this
   License. However, in accepting such obligations, You may act only
   on Your own behalf and on Your sole responsibility, not on behalf
   of any other Contributor, and only if You agree to indemnify,
   defend, and hold each Contributor harmless for any liability
   incurred by, or claims asserted against, such Contributor by reason
   of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS
```

C.3.13 expat

The `pyexpat` extension is built using an included copy of the expat sources unless the build is configured `--with-system-expat`:

```
Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd
                               and Clark Cooper

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.14 libffi

The `_ctypes` C extension underlying the `ctypes` module is built using an included copy of the libffi sources unless the build is configured `--with-system-libffi`:

```
Copyright (c) 1996-2008  Red Hat, Inc and others.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
``Software''), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED ``AS IS'', WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT.  IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

C.3.15 zlib

The `zlib` extension is built using an included copy of the zlib sources if the zlib version found on the system is too old to be used for the build:

```
Copyright (C) 1995-2011 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied
warranty.  In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.

3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly        Mark Adler
jloup@gzip.org          madler@alumni.caltech.edu
```

C.3.16 cfuhash

The implementation of the hash table used by the `tracemalloc` is based on the cfuhash project:

```
Copyright (c) 2005 Don Owens
All rights reserved.

This code is released under the BSD license:

Redistribution and use in source and binary forms, with or without
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
modification, are permitted provided that the following conditions
are met:

  * Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

  * Redistributions in binary form must reproduce the above
    copyright notice, this list of conditions and the following
    disclaimer in the documentation and/or other materials provided
    with the distribution.

  * Neither the name of the author nor the names of its
    contributors may be used to endorse or promote products derived
    from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
OF THE POSSIBILITY OF SUCH DAMAGE.
```

C.3.17 libmpdec

The `_decimal` C extension underlying the `decimal` module is built using an included copy of the libmpdec library unless the build is configured `--with-system-libmpdec`:

```
Copyright (c) 2008-2020 Stefan Krah. All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
SUCH DAMAGE.
```

C.3.18 W3C C14N test suite

The C14N 2.0 test suite in the `test` package (`Lib/test/xmltestdata/c14n-20/`) was retrieved from the W3C website at https://www.w3.org/TR/xml-c14n2-testcases/ and is distributed under the 3-clause BSD license:

```
Copyright (c) 2013 W3C(R) (MIT, ERCIM, Keio, Beihang),
All Rights Reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

* Redistributions of works must retain the original copyright notice,
  this list of conditions and the following disclaimer.
* Redistributions in binary form must reproduce the original copyright
  notice, this list of conditions and the following disclaimer in the
  documentation and/or other materials provided with the distribution.
* Neither the name of the W3C nor the names of its contributors may be
  used to endorse or promote products derived from this work without
  specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

C.3.19 mimalloc

MIT License:

```
Copyright (c) 2018-2021 Microsoft Corporation, Daan Leijen

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

C.3.20 asyncio

Parts of the `asyncio` module are incorporated from uvloop 0.16, which is distributed under the MIT license:

```
Copyright (c) 2015-2021 MagicStack Inc.  http://magic.io

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.21 Global Unbounded Sequences (GUS)

The file `Python/qsbr.c` is adapted from FreeBSD's „Global Unbounded Sequences" safe memory reclamation scheme in subr_smr.c. The file is distributed under the 2-Clause BSD License:

```
Copyright (c) 2019,2020 Jeffrey Roberson <jeff@FreeBSD.org>

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice unmodified, this list of conditions, and the following
   disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS'' AND ANY EXPRESS OR
IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF
THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

DODATEK D

Prawa autorskie

Python i ta dokumentacja jest:

Copyright © 2001-2024 Python Software Foundation. All rights reserved.

Copyright © 2000 BeOpen.com. Wszystkie prawa zastrzeżone.

Copyright © 1995-2000 Corporation for National Research Initiatives. Wszystkie prawa zastrzeżone.

Copyright © 1991-1995 Stichting Mathematisch Centrum. Wszystkie prawa zastrzeżone.

Patrz dział *Historia i zapisy prawne*, aby zobaczyć pełną informację na temat licencji i praw.

Indeks

Niealfabetyczny

. . . , **321**
>>>, **321**
__all__ (*package variable*), 70
__dict__ (*module attribute*), 176
__doc__ (*module attribute*), 176
__file__ (*module attribute*), 176, 177
__future__, **327**
__import__
    funkcja wbudowana, 70
__loader__ (*module attribute*), 176
__main__
    moduł, 12, 205, 219, 220
__name__ (*module attribute*), 176
__package__ (*module attribute*), 176
__PYVENV_LAUNCHER__, 236, 242
__slots__, **334**
_frozen (*C struct*), 73
_inittab (*C struct*), 73
_inittab.initfunc (*C member*), 73
_inittab.name (*C member*), 73
_Py_c_diff (*C function*), 135
_Py_c_neg (*C function*), 135
_Py_c_pow (*C function*), 135
_Py_c_prod (*C function*), 135
_Py_c_quot (*C function*), 135
_Py_c_sum (*C function*), 135
_Py_InitializeMain (*C function*), 249
_Py_NoneStruct (*C var*), 264
_PyBytes_Resize (*C function*), 138
_PyCode_GetExtra (*C function*), 174
_PyCode_SetExtra (*C function*), 175
_PyEval_RequestCodeExtraIndex (*C function*), 174
_PyFrameEvalFunction (*C type*), 216
_PyInterpreterFrame (*C struct*), 192
_PyInterpreterState_GetEvalFrameFunc (*C function*), 217
_PyInterpreterState_SetEvalFrameFunc (*C function*), 217
_PyObject_GetDictPtr (*C function*), 97
_PyObject_New (*C function*), 263
_PyObject_NewVar (*C function*), 263
_PyTuple_Resize (*C function*), 158
_thread
    moduł, 213

A

abort (*C function*), 70
abs
    funkcja wbudowana, 105
abstrakcyjna klasa bazowa, **321**
adnotacja, **321**
allocfunc (*C type*), 304
argument, **321**
argv (*in module sys*), 210, 236
ascii
    funkcja wbudowana, 97
asynchronous context manager, **322**
asynchronous generator, **322**
asynchronous generator iterator, **322**
asynchronous iterable, **322**
asynchronous iterator, **322**
atrybut, **322**
awaitable, **322**

B

BDFL, **322**
binary file, **323**
binaryfunc (*C type*), 305
borrowed reference, **323**
buffer interface
    (see buffer protocol), 112
buffer object
    (see buffer protocol), 112
buffer protocol, 112
builtins
    moduł, 12, 205, 219, 220
bytearray
    obiekt, 138
bytes
    funkcja wbudowana, 98
    obiekt, 136
bytes-like object, **323**

C

callable, **323**

calloc (*C function*), 251
Capsule
    obiekt, 189
C-contiguous, 115, 324
class, **323**
class variable, **323**
classmethod
    funkcja wbudowana, 268
cleanup functions, 70
close (*in module os*), 219
closure variable, **323**
CO_FUTURE_DIVISION (*C var*), 46
code object, 171
Common Vulnerabilities and Exposures
    CVE 2008-5983, 210
compile
    funkcja wbudowana, 71
context manager, **324**
context variable, **324**
contiguous, 115, **324**
copyright (*in module sys*), 209
coroutine, **324**
coroutine function, **324**
CPython, **324**

D

decorator, **324**
descrgetfunc (*C type*), 304
descriptor, **325**
descrsetfunc (*C type*), 304
destructor (*C type*), 304
dictionary comprehension, **325**
dictionary view, **325**
divmod
    funkcja wbudowana, 105
docstring, **325**
duck-typing, **325**

E

EAFP, **325**
EOFError (*built-in exception*), 175
exc_info (*in module sys*), 11
executable (*in module sys*), 209
exit (*C function*), 70
expression, **325**

F

f-string, **326**
file object, **326**
file-like object, **326**
filesystem encoding and error handler, **326**
finder, **326**
float
    funkcja wbudowana, 107
floating-point
    obiekt, 133
floor division, **326**
Fortran contiguous, 115, 324
free (*C function*), 251
free threading, **326**
free variable, **326**
freefunc (*C type*), 304
freeze utility, 73
frozenset
    obiekt, 166
function annotation, **326**
funkcja, **326**
    obiekt, 168
funkcja wbudowana
    __import__, 70
    abs, 105
    ascii, 97
    bytes, 98
    classmethod, 268
    compile, 71
    divmod, 105
    float, 107
    hash, 98, 282
    int, 107
    len, 99, 108, 110, 160, 164, 167
    pow, 105, 107
    repr, 97, 281
    staticmethod, 268
    tuple, 109, 161
    type, 98

G

garbage collection, **327**
gcvisitobjects_t (*C type*), 311
generator, **327**
generator expression, **327**
generator iterator, **327**
generic function, **327**
generic type, **327**
getattrfunc (*C type*), 304
getattrofunc (*C type*), 304
getbufferproc (*C type*), 305
getiterfunc (*C type*), 304
getter (*C type*), 272
GIL, **327**
global interpreter lock, 211, **327**

H

hash
    funkcja wbudowana, 98, 282
hash-based pyc, **328**
hashable, **328**
hashfunc (*C type*), 304

I

IDLE, **328**
immortal, **328**
immutable, **328**
import path, **328**
importer, **328**
importing, **328**

incr_item(), 11, 12
initproc (*C type*), 304
inquiry (*C type*), 310
instancemethod
    obiekt, 170
instrukcja, **335**
int
    funkcja wbudowana, 107
integer
    obiekt, 125
interactive, **328**
interpreted, **328**
interpreter lock, 211
interpreter shutdown, **329**
iterable, **329**
iterator, **329**
iternextfunc (*C type*), 304

K

key function, **329**
KeyboardInterrupt (*built-in exception*), 59
keyword argument, **329**
kod bajtowy, **323**

L

lambda, **329**
LBYL, **329**
len
    funkcja wbudowana, 99, 108, 110, 160, 164, 167
lenfunc (*C type*), 304
liczba zespolona, **324**
    obiekt, 134
list, **330**
    obiekt, 159
list comprehension, **330**
loader, **330**
locale encoding, **330**
lock, interpreter, 211
long integer
    obiekt, 125
LONG_MAX (*C macro*), 127

M

magic
    method, 330
magic method, **330**
main(), 207, 210, 236
malloc (*C function*), 251
mapping, **330**
    obiekt, 161
memoryview
    obiekt, 186
meta path finder, **330**
metaclass, **330**
METH_CLASS (*C macro*), 268
METH_COEXIST (*C macro*), 268
METH_FASTCALL (*C macro*), 267
METH_KEYWORDS (*C macro*), 267
METH_METHOD (*C macro*), 267
METH_NOARGS (*C macro*), 267
METH_O (*C macro*), 268
METH_STATIC (*C macro*), 268
METH_VARARGS (*C macro*), 267
method, **330**
    magic, 330
    obiekt, 170
    special, 335
method resolution order, **331**
MethodType (*in module types*), 168, 170
module spec, **331**
modules (*in module sys*), 70, 205
ModuleType (*in module types*), 176
moduł, **331**
    __main__, 12, 205, 219, 220
    _thread, 213
    builtins, 12, 205, 219, 220
    obiekt, 176
    signal, 59
    sys, 12, 205, 219, 220
    wyszukiwanie ścieżka, 12, 205, 209
moduł rozszerzenia, **325**
MRO, **331**
mutable, **331**

N

named tuple, **331**
namespace, **331**
namespace package, **331**
nested scope, **331**
new-style class, **331**
newfunc (*C type*), 304
None
    obiekt, 125
numeric
    obiekt, 125

O

obiekt, **332**
    bytearray, 138
    bytes, 136
    Capsule, 189
    code, 171
    floating-point, 133
    frozenset, 166
    funkcja, 168
    instancemethod, 170
    integer, 125
    liczba zespolona, 134
    list, 159
    long integer, 125
    mapping, 161
    memoryview, 186
    method, 170
    moduł, 176
    None, 125

numeric, 125
plik, 175
sequence, 136
set, 166
słownik, 161
tuple, 156
type, 7, 119
objobjargproc (*C type*), 305
objobjproc (*C type*), 305
optimized scope, **332**
OverflowError (*built-in exception*), 127, 128

P

package, **332**
package variable
__all__, 70
parameter, **332**
PATH, 12
path (*in module sys*), 12, 205, 209
path based finder, **333**
path entry, **332**
path entry finder, **332**
path entry hook, **333**
path-like object, **333**
PEP, **333**
platform (*in module sys*), 209
plik
obiekt, 175
portion, **333**
positional argument, **333**
pow
funkcja wbudowana, 105, 107
provisional API, **333**
provisional package, **333**
Py_ABS (*C macro*), 4
Py_AddPendingCall (*C function*), 221
Py_ALWAYS_INLINE (*C macro*), 5
Py_ASNATIVEBYTES_ALLOW_INDEX (*C macro*), 131
Py_ASNATIVEBYTES_BIG_ENDIAN (*C macro*), 131
Py_ASNATIVEBYTES_DEFAULTS (*C macro*), 131
Py_ASNATIVEBYTES_LITTLE_ENDIAN (*C macro*), 131
Py_ASNATIVEBYTES_NATIVE_ENDIAN (*C macro*), 131
Py_ASNATIVEBYTES_REJECT_NEGATIVE (*C macro*), 131
Py_ASNATIVEBYTES_UNSIGNED_BUFFER (*C macro*), 131
Py_AtExit (*C function*), 70
Py_AUDIT_READ (*C macro*), 270
Py_AuditHookFunction (*C type*), 69
Py_BEGIN_ALLOW_THREADS (*C macro*), 211, 214
Py_BEGIN_CRITICAL_SECTION (*C macro*), 227
Py_BEGIN_CRITICAL_SECTION2 (*C macro*), 227
Py_BLOCK_THREADS (*C macro*), 215
Py_buffer (*C type*), 113
Py_buffer.buf (*C member*), 113
Py_buffer.format (*C member*), 113
Py_buffer.internal (*C member*), 114
Py_buffer.itemsize (*C member*), 113
Py_buffer.len (*C member*), 113
Py_buffer.ndim (*C member*), 113
Py_buffer.obj (*C member*), 113
Py_buffer.readonly (*C member*), 113
Py_buffer.shape (*C member*), 114
Py_buffer.strides (*C member*), 114
Py_buffer.suboffsets (*C member*), 114
Py_BuildValue (*C function*), 81
Py_BytesMain (*C function*), 206
Py_BytesWarningFlag (*C var*), 202
Py_CHARMASK (*C macro*), 5
Py_CLEAR (*C function*), 48
Py_CompileString (*C function*), 45, 46
Py_CompileStringExFlags (*C function*), 45
Py_CompileStringFlags (*C function*), 45
Py_CompileStringObject (*C function*), 45
Py_complex (*C type*), 134
Py_complex.imag (*C member*), 134
Py_complex.real (*C member*), 134
Py_CONSTANT_ELLIPSIS (*C macro*), 94
Py_CONSTANT_EMPTY_BYTES (*C macro*), 94
Py_CONSTANT_EMPTY_STR (*C macro*), 94
Py_CONSTANT_EMPTY_TUPLE (*C macro*), 94
Py_CONSTANT_FALSE (*C macro*), 94
Py_CONSTANT_NONE (*C macro*), 94
Py_CONSTANT_NOT_IMPLEMENTED (*C macro*), 94
Py_CONSTANT_ONE (*C macro*), 94
Py_CONSTANT_TRUE (*C macro*), 94
Py_CONSTANT_ZERO (*C macro*), 94
PY_CXX_CONST (*C macro*), 81
Py_DEBUG (*C macro*), 13
Py_DebugFlag (*C var*), 202
Py_DecodeLocale (*C function*), 66
Py_DECREF (*C function*), 7, 48
Py_DecRef (*C function*), 49
Py_DEPRECATED (*C macro*), 5
Py_DontWriteBytecodeFlag (*C var*), 203
Py_Ellipsis (*C var*), 186
Py_EncodeLocale (*C function*), 67
Py_END_ALLOW_THREADS (*C macro*), 211, 215
Py_END_CRITICAL_SECTION (*C macro*), 227
Py_END_CRITICAL_SECTION2 (*C macro*), 228
Py_EndInterpreter (*C function*), 220
Py_EnterRecursiveCall (*C function*), 62
Py_EQ (*C macro*), 291
Py_eval_input (*C var*), 46
Py_Exit (*C function*), 70
Py_ExitStatusException (*C function*), 231
Py_False (*C var*), 132
Py_FatalError (*C function*), 70
Py_FatalError(), 210
Py_FdIsInteractive (*C function*), 65
Py_file_input (*C var*), 46
Py_Finalize (*C function*), 206
Py_FinalizeEx (*C function*), 70, 205, 206, 219, 220
Py_FrozenFlag (*C var*), 203

Py_GE (*C macro*), 291
Py_GenericAlias (*C function*), 198
Py_GenericAliasType (*C var*), 199
Py_GetArgcArgv (*C function*), 249
Py_GetBuildInfo (*C function*), 209
Py_GetCompiler (*C function*), 209
Py_GetConstant (*C function*), 93
Py_GetConstantBorrowed (*C function*), 94
Py_GetCopyright (*C function*), 209
Py_GETENV (*C macro*), 5
Py_GetExecPrefix (*C function*), 12, 208
Py_GetPath (*C function*), 12, 209
Py_GetPath(), 207
Py_GetPlatform (*C function*), 209
Py_GetPrefix (*C function*), 12, 208
Py_GetProgramFullPath (*C function*), 12, 208
Py_GetProgramName (*C function*), 208
Py_GetPythonHome (*C function*), 211
Py_GetVersion (*C function*), 209
Py_GT (*C macro*), 291
Py_hash_t (*C type*), 85
Py_HashPointer (*C function*), 86
Py_HashRandomizationFlag (*C var*), 203
Py_IgnoreEnvironmentFlag (*C var*), 203
Py_INCREF (*C function*), 7, 47
Py_IncRef (*C function*), 49
Py_Initialize (*C function*), 12, 205, 219
Py_Initialize(), 207
Py_InitializeEx (*C function*), 205
Py_InitializeFromConfig (*C function*), 205
Py_InspectFlag (*C var*), 203
Py_InteractiveFlag (*C var*), 203
Py_Is (*C function*), 264
Py_IS_TYPE (*C function*), 265
Py_IsFalse (*C function*), 265
Py_IsFinalizing (*C function*), 206
Py_IsInitialized (*C function*), 12, 206
Py_IsNone (*C function*), 264
Py_IsolatedFlag (*C var*), 203
Py_IsTrue (*C function*), 265
Py_LE (*C macro*), 291
Py_LeaveRecursiveCall (*C function*), 62
Py_LegacyWindowsFSEncodingFlag (*C var*), 204
Py_LegacyWindowsStdioFlag (*C var*), 204
Py_LIMITED_API (*C macro*), 16
Py_LT (*C macro*), 291
Py_Main (*C function*), 206
PY_MAJOR_VERSION (*C macro*), 313
Py_MAX (*C macro*), 5
Py_MEMBER_SIZE (*C macro*), 5
PY_MICRO_VERSION (*C macro*), 313
Py_MIN (*C macro*), 5
PY_MINOR_VERSION (*C macro*), 313
Py_mod_create (*C macro*), 179
Py_mod_exec (*C macro*), 180
Py_mod_gil (*C macro*), 180
Py_MOD_GIL_NOT_USED (*C macro*), 180
Py_MOD_GIL_USED (*C macro*), 180
Py_mod_multiple_interpreters (*C macro*), 180
Py_MOD_MULTIPLE_INTERPRETERS_NOT_SUPPORTED (*C macro*), 180
Py_MOD_MULTIPLE_INTERPRETERS_SUPPORTED (*C macro*), 180
Py_MOD_PER_INTERPRETER_GIL_SUPPORTED (*C macro*), 180
PY_MONITORING_EVENT_BRANCH (*C macro*), 319
PY_MONITORING_EVENT_C_RAISE (*C macro*), 319
PY_MONITORING_EVENT_C_RETURN (*C macro*), 319
PY_MONITORING_EVENT_CALL (*C macro*), 319
PY_MONITORING_EVENT_EXCEPTION_HANDLED (*C macro*), 319
PY_MONITORING_EVENT_INSTRUCTION (*C macro*), 319
PY_MONITORING_EVENT_JUMP (*C macro*), 319
PY_MONITORING_EVENT_LINE (*C macro*), 319
PY_MONITORING_EVENT_PY_RESUME (*C macro*), 319
PY_MONITORING_EVENT_PY_RETURN (*C macro*), 319
PY_MONITORING_EVENT_PY_START (*C macro*), 319
PY_MONITORING_EVENT_PY_THROW (*C macro*), 319
PY_MONITORING_EVENT_PY_UNWIND (*C macro*), 319
PY_MONITORING_EVENT_PY_YIELD (*C macro*), 319
PY_MONITORING_EVENT_RAISE (*C macro*), 319
PY_MONITORING_EVENT_RERAISE (*C macro*), 319
PY_MONITORING_EVENT_STOP_ITERATION (*C macro*), 319
Py_NE (*C macro*), 291
Py_NewInterpreter (*C function*), 219
Py_NewInterpreterFromConfig (*C function*), 219
Py_NewRef (*C function*), 48
Py_NO_INLINE (*C macro*), 5
Py_None (*C var*), 125
Py_NoSiteFlag (*C var*), 204
Py_NotImplemented (*C var*), 94
Py_NoUserSiteDirectory (*C var*), 204
Py_OpenCodeHookFunction (*C type*), 175
Py_OptimizeFlag (*C var*), 204
Py_PreInitialize (*C function*), 234
Py_PreInitializeFromArgs (*C function*), 234
Py_PreInitializeFromBytesArgs (*C function*), 234
Py_PRINT_RAW (*C macro*), 94, 176
Py_QuietFlag (*C var*), 205
Py_READONLY (*C macro*), 270
Py_REFCNT (*C function*), 47
Py_RELATIVE_OFFSET (*C macro*), 270
PY_RELEASE_LEVEL (*C macro*), 313
PY_RELEASE_SERIAL (*C macro*), 313
Py_ReprEnter (*C function*), 62
Py_ReprLeave (*C function*), 62
Py_RETURN_FALSE (*C macro*), 132
Py_RETURN_NONE (*C macro*), 125
Py_RETURN_NOTIMPLEMENTED (*C macro*), 94
Py_RETURN_RICHCOMPARE (*C macro*), 291
Py_RETURN_TRUE (*C macro*), 132
Py_RunMain (*C function*), 207
Py_SET_REFCNT (*C function*), 47

Py_SET_SIZE (*C function*), 265
Py_SET_TYPE (*C function*), 265
Py_SetProgramName (*C function*), 207
Py_SetPythonHome (*C function*), 210
Py_SETREF (*C macro*), 49
Py_single_input (*C var*), 46
Py_SIZE (*C function*), 265
Py_ssize_t (*C type*), 10
PY_SSIZE_T_MAX (*C macro*), 128
Py_STRINGIFY (*C macro*), 5
Py_T_BOOL (*C macro*), 271
Py_T_BYTE (*C macro*), 271
Py_T_CHAR (*C macro*), 271
Py_T_DOUBLE (*C macro*), 271
Py_T_FLOAT (*C macro*), 271
Py_T_INT (*C macro*), 271
Py_T_LONG (*C macro*), 271
Py_T_LONGLONG (*C macro*), 271
Py_T_OBJECT_EX (*C macro*), 271
Py_T_PYSSIZET (*C macro*), 271
Py_T_SHORT (*C macro*), 271
Py_T_STRING (*C macro*), 271
Py_T_STRING_INPLACE (*C macro*), 271
Py_T_UBYTE (*C macro*), 271
Py_T_UINT (*C macro*), 271
Py_T_ULONG (*C macro*), 271
Py_T_ULONGLONG (*C macro*), 271
Py_T_USHORT (*C macro*), 271
Py_TPFLAGS_BASE_EXC_SUBCLASS (*C macro*), 286
Py_TPFLAGS_BASETYPE (*C macro*), 284
Py_TPFLAGS_BYTES_SUBCLASS (*C macro*), 286
Py_TPFLAGS_DEFAULT (*C macro*), 285
Py_TPFLAGS_DICT_SUBCLASS (*C macro*), 286
Py_TPFLAGS_DISALLOW_INSTANTIATION (*C macro*), 287
Py_TPFLAGS_HAVE_FINALIZE (*C macro*), 286
Py_TPFLAGS_HAVE_GC (*C macro*), 285
Py_TPFLAGS_HAVE_VECTORCALL (*C macro*), 286
Py_TPFLAGS_HEAPTYPE (*C macro*), 284
Py_TPFLAGS_IMMUTABLETYPE (*C macro*), 286
Py_TPFLAGS_ITEMS_AT_END (*C macro*), 286
Py_TPFLAGS_LIST_SUBCLASS (*C macro*), 286
Py_TPFLAGS_LONG_SUBCLASS (*C macro*), 286
Py_TPFLAGS_MANAGED_DICT (*C macro*), 285
Py_TPFLAGS_MANAGED_WEAKREF (*C macro*), 285
Py_TPFLAGS_MAPPING (*C macro*), 287
Py_TPFLAGS_METHOD_DESCRIPTOR (*C macro*), 285
Py_TPFLAGS_READY (*C macro*), 284
Py_TPFLAGS_READYING (*C macro*), 285
Py_TPFLAGS_SEQUENCE (*C macro*), 287
Py_TPFLAGS_TUPLE_SUBCLASS (*C macro*), 286
Py_TPFLAGS_TYPE_SUBCLASS (*C macro*), 286
Py_TPFLAGS_UNICODE_SUBCLASS (*C macro*), 286
Py_TPFLAGS_VALID_VERSION_TAG (*C macro*), 288
Py_tracefunc (*C type*), 221
Py_True (*C var*), 132
Py_tss_NEEDS_INIT (*C macro*), 224
Py_tss_t (*C type*), 224
Py_TYPE (*C function*), 265
Py_UCS1 (*C type*), 139
Py_UCS2 (*C type*), 139
Py_UCS4 (*C type*), 139
Py_uhash_t (*C type*), 85
Py_UNBLOCK_THREADS (*C macro*), 215
Py_UnbufferedStdioFlag (*C var*), 205
Py_UNICODE (*C type*), 139
Py_UNICODE_IS_HIGH_SURROGATE (*C function*), 142
Py_UNICODE_IS_LOW_SURROGATE (*C function*), 142
Py_UNICODE_IS_SURROGATE (*C function*), 142
Py_UNICODE_ISALNUM (*C function*), 142
Py_UNICODE_ISALPHA (*C function*), 142
Py_UNICODE_ISDECIMAL (*C function*), 141
Py_UNICODE_ISDIGIT (*C function*), 141
Py_UNICODE_ISLINEBREAK (*C function*), 141
Py_UNICODE_ISLOWER (*C function*), 141
Py_UNICODE_ISNUMERIC (*C function*), 141
Py_UNICODE_ISPRINTABLE (*C function*), 142
Py_UNICODE_ISSPACE (*C function*), 141
Py_UNICODE_ISTITLE (*C function*), 141
Py_UNICODE_ISUPPER (*C function*), 141
Py_UNICODE_JOIN_SURROGATES (*C function*), 142
Py_UNICODE_TODECIMAL (*C function*), 142
Py_UNICODE_TODIGIT (*C function*), 142
Py_UNICODE_TOLOWER (*C function*), 142
Py_UNICODE_TONUMERIC (*C function*), 142
Py_UNICODE_TOTITLE (*C function*), 142
Py_UNICODE_TOUPPER (*C function*), 142
Py_UNREACHABLE (*C macro*), 6
Py_UNUSED (*C macro*), 6
Py_VaBuildValue (*C function*), 83
PY_VECTORCALL_ARGUMENTS_OFFSET (*C macro*), 101
Py_VerboseFlag (*C var*), 205
Py_Version (*C var*), 313
PY_VERSION_HEX (*C macro*), 313
Py_VISIT (*C function*), 309
Py_XDECREF (*C function*), 12, 48
Py_XINCREF (*C function*), 47
Py_XNewRef (*C function*), 48
Py_XSETREF (*C macro*), 49
PyAIter_Check (*C function*), 111
PyAnySet_Check (*C function*), 166
PyAnySet_CheckExact (*C function*), 167
PyArg_Parse (*C function*), 80
PyArg_ParseTuple (*C function*), 79
PyArg_ParseTupleAndKeywords (*C function*), 79
PyArg_UnpackTuple (*C function*), 80
PyArg_ValidateKeywordArguments (*C function*), 80
PyArg_VaParse (*C function*), 79
PyArg_VaParseTupleAndKeywords (*C function*), 80
PyASCIIObject (*C type*), 139
PyAsyncMethods (*C type*), 303
PyAsyncMethods.am_aiter (*C member*), 303
PyAsyncMethods.am_anext (*C member*), 303
PyAsyncMethods.am_await (*C member*), 303

PyAsyncMethods.am_send (*C member*), 303
PyBool_Check (*C function*), 132
PyBool_FromLong (*C function*), 132
PyBool_Type (*C var*), 132
PyBUF_ANY_CONTIGUOUS (*C macro*), 115
PyBUF_C_CONTIGUOUS (*C macro*), 115
PyBUF_CONTIG (*C macro*), 116
PyBUF_CONTIG_RO (*C macro*), 116
PyBUF_F_CONTIGUOUS (*C macro*), 115
PyBUF_FORMAT (*C macro*), 115
PyBUF_FULL (*C macro*), 116
PyBUF_FULL_RO (*C macro*), 116
PyBUF_INDIRECT (*C macro*), 115
PyBUF_MAX_NDIM (*C macro*), 114
PyBUF_ND (*C macro*), 115
PyBUF_READ (*C macro*), 187
PyBUF_RECORDS (*C macro*), 116
PyBUF_RECORDS_RO (*C macro*), 116
PyBUF_SIMPLE (*C macro*), 115
PyBUF_STRIDED (*C macro*), 116
PyBUF_STRIDED_RO (*C macro*), 116
PyBUF_STRIDES (*C macro*), 115
PyBUF_WRITABLE (*C macro*), 115
PyBUF_WRITE (*C macro*), 187
PyBuffer_FillContiguousStrides (*C function*), 118
PyBuffer_FillInfo (*C function*), 118
PyBuffer_FromContiguous (*C function*), 118
PyBuffer_GetPointer (*C function*), 118
PyBuffer_IsContiguous (*C function*), 118
PyBuffer_Release (*C function*), 118
PyBuffer_SizeFromFormat (*C function*), 118
PyBuffer_ToContiguous (*C function*), 118
PyBufferProcs (*C type*), 112, 302
PyBufferProcs.bf_getbuffer (*C member*), 302
PyBufferProcs.bf_releasebuffer (*C member*), 302
PyByteArray_AS_STRING (*C function*), 139
PyByteArray_AsString (*C function*), 139
PyByteArray_Check (*C function*), 138
PyByteArray_CheckExact (*C function*), 138
PyByteArray_Concat (*C function*), 138
PyByteArray_FromObject (*C function*), 138
PyByteArray_FromStringAndSize (*C function*), 138
PyByteArray_GET_SIZE (*C function*), 139
PyByteArray_Resize (*C function*), 139
PyByteArray_Size (*C function*), 139
PyByteArray_Type (*C var*), 138
PyByteArrayObject (*C type*), 138
PyBytes_AS_STRING (*C function*), 137
PyBytes_AsString (*C function*), 137
PyBytes_AsStringAndSize (*C function*), 137
PyBytes_Check (*C function*), 136
PyBytes_CheckExact (*C function*), 136
PyBytes_Concat (*C function*), 138
PyBytes_ConcatAndDel (*C function*), 138
PyBytes_FromFormat (*C function*), 136
PyBytes_FromFormatV (*C function*), 137
PyBytes_FromObject (*C function*), 137
PyBytes_FromString (*C function*), 136
PyBytes_FromStringAndSize (*C function*), 136
PyBytes_GET_SIZE (*C function*), 137
PyBytes_Size (*C function*), 137
PyBytes_Type (*C var*), 136
PyBytesObject (*C type*), 136
PyCallable_Check (*C function*), 104
PyCallIter_Check (*C function*), 184
PyCallIter_New (*C function*), 184
PyCallIter_Type (*C var*), 184
PyCapsule (*C type*), 189
PyCapsule_CheckExact (*C function*), 189
PyCapsule_Destructor (*C type*), 189
PyCapsule_GetContext (*C function*), 189
PyCapsule_GetDestructor (*C function*), 189
PyCapsule_GetName (*C function*), 189
PyCapsule_GetPointer (*C function*), 189
PyCapsule_Import (*C function*), 190
PyCapsule_IsValid (*C function*), 190
PyCapsule_New (*C function*), 189
PyCapsule_SetContext (*C function*), 190
PyCapsule_SetDestructor (*C function*), 190
PyCapsule_SetName (*C function*), 190
PyCapsule_SetPointer (*C function*), 190
PyCell_Check (*C function*), 171
PyCell_GET (*C function*), 171
PyCell_Get (*C function*), 171
PyCell_New (*C function*), 171
PyCell_SET (*C function*), 171
PyCell_Set (*C function*), 171
PyCell_Type (*C var*), 171
PyCellObject (*C type*), 171
PyCFunction (*C type*), 266
PyCFunction_New (*C function*), 268
PyCFunction_NewEx (*C function*), 268
PyCFunctionFast (*C type*), 266
PyCFunctionFastWithKeywords (*C type*), 266
PyCFunctionWithKeywords (*C type*), 266
PyCMethod (*C type*), 266
PyCMethod_New (*C function*), 268
PyCode_Addr2Line (*C function*), 172
PyCode_Addr2Location (*C function*), 172
PyCode_AddWatcher (*C function*), 173
PyCode_Check (*C function*), 171
PyCode_ClearWatcher (*C function*), 173
PyCode_GetCellvars (*C function*), 173
PyCode_GetCode (*C function*), 173
PyCode_GetFreevars (*C function*), 173
PyCode_GetNumFree (*C function*), 171
PyCode_GetVarnames (*C function*), 173
PyCode_New (*C function*), 172
PyCode_NewEmpty (*C function*), 172
PyCode_NewWithPosOnlyArgs (*C function*), 172
PyCode_Type (*C var*), 171
PyCode_WatchCallback (*C type*), 173

PyCodec_BackslashReplaceErrors (*C function*), 89
PyCodec_Decode (*C function*), 87
PyCodec_Decoder (*C function*), 88
PyCodec_Encode (*C function*), 87
PyCodec_Encoder (*C function*), 88
PyCodec_IgnoreErrors (*C function*), 88
PyCodec_IncrementalDecoder (*C function*), 88
PyCodec_IncrementalEncoder (*C function*), 88
PyCodec_KnownEncoding (*C function*), 87
PyCodec_LookupError (*C function*), 88
PyCodec_NameReplaceErrors (*C function*), 89
PyCodec_Register (*C function*), 87
PyCodec_RegisterError (*C function*), 88
PyCodec_ReplaceErrors (*C function*), 88
PyCodec_StreamReader (*C function*), 88
PyCodec_StreamWriter (*C function*), 88
PyCodec_StrictErrors (*C function*), 88
PyCodec_Unregister (*C function*), 87
PyCodec_XMLCharRefReplaceErrors (*C function*), 89
PyCodeEvent (*C type*), 173
PyCodeObject (*C type*), 171
PyCompactUnicodeObject (*C type*), 139
PyCompilerFlags (*C struct*), 46
PyCompilerFlags.cf_feature_version (*C member*), 46
PyCompilerFlags.cf_flags (*C member*), 46
PyComplex_AsCComplex (*C function*), 136
PyComplex_Check (*C function*), 135
PyComplex_CheckExact (*C function*), 135
PyComplex_FromCComplex (*C function*), 135
PyComplex_FromDoubles (*C function*), 135
PyComplex_ImagAsDouble (*C function*), 136
PyComplex_RealAsDouble (*C function*), 135
PyComplex_Type (*C var*), 135
PyComplexObject (*C type*), 135
PyConfig (*C type*), 234
PyConfig_Clear (*C function*), 235
PyConfig_InitIsolatedConfig (*C function*), 235
PyConfig_InitPythonConfig (*C function*), 235
PyConfig_Read (*C function*), 235
PyConfig_SetArgv (*C function*), 235
PyConfig_SetBytesArgv (*C function*), 235
PyConfig_SetBytesString (*C function*), 235
PyConfig_SetString (*C function*), 235
PyConfig_SetWideStringList (*C function*), 235
PyConfig.argv (*C member*), 236
PyConfig.base_exec_prefix (*C member*), 236
PyConfig.base_executable (*C member*), 236
PyConfig.base_prefix (*C member*), 236
PyConfig.buffered_stdio (*C member*), 237
PyConfig.bytes_warning (*C member*), 237
PyConfig.check_hash_pycs_mode (*C member*), 237
PyConfig.code_debug_ranges (*C member*), 237
PyConfig.configure_c_stdio (*C member*), 237
PyConfig.cpu_count (*C member*), 240
PyConfig.dev_mode (*C member*), 237
PyConfig.dump_refs (*C member*), 238
PyConfig.exec_prefix (*C member*), 238
PyConfig.executable (*C member*), 238
PyConfig.faulthandler (*C member*), 238
PyConfig.filesystem_encoding (*C member*), 238
PyConfig.filesystem_errors (*C member*), 238
PyConfig.hash_seed (*C member*), 239
PyConfig.home (*C member*), 239
PyConfig.import_time (*C member*), 239
PyConfig.inspect (*C member*), 239
PyConfig.install_signal_handlers (*C member*), 239
PyConfig.int_max_str_digits (*C member*), 239
PyConfig.interactive (*C member*), 239
PyConfig.isolated (*C member*), 240
PyConfig.legacy_windows_stdio (*C member*), 240
PyConfig.malloc_stats (*C member*), 240
PyConfig.module_search_paths (*C member*), 241
PyConfig.module_search_paths_set (*C member*), 241
PyConfig.optimization_level (*C member*), 241
PyConfig.orig_argv (*C member*), 241
PyConfig.parse_argv (*C member*), 241
PyConfig.parser_debug (*C member*), 242
PyConfig.pathconfig_warnings (*C member*), 242
PyConfig.perf_profiling (*C member*), 244
PyConfig.platlibdir (*C member*), 240
PyConfig.prefix (*C member*), 242
PyConfig.program_name (*C member*), 242
PyConfig.pycache_prefix (*C member*), 242
PyConfig.pythonpath_env (*C member*), 241
PyConfig.quiet (*C member*), 242
PyConfig.run_command (*C member*), 242
PyConfig.run_filename (*C member*), 242
PyConfig.run_module (*C member*), 243
PyConfig.run_presite (*C member*), 243
PyConfig.safe_path (*C member*), 236
PyConfig.show_ref_count (*C member*), 243
PyConfig.site_import (*C member*), 243
PyConfig.skip_source_first_line (*C member*), 243
PyConfig.stdio_encoding (*C member*), 243
PyConfig.stdio_errors (*C member*), 243
PyConfig.tracemalloc (*C member*), 244
PyConfig.use_environment (*C member*), 244
PyConfig.use_hash_seed (*C member*), 239
PyConfig.user_site_directory (*C member*), 244
PyConfig.verbose (*C member*), 244
PyConfig.warn_default_encoding (*C member*), 237
PyConfig.warnoptions (*C member*), 244
PyConfig.write_bytecode (*C member*), 245
PyConfig.xoptions (*C member*), 245
PyContext (*C type*), 194
PyContext_CheckExact (*C function*), 194
PyContext_Copy (*C function*), 194

PyContext_CopyCurrent (*C function*), 194
PyContext_Enter (*C function*), 194
PyContext_Exit (*C function*), 194
PyContext_New (*C function*), 194
PyContext_Type (*C var*), 194
PyContextToken (*C type*), 194
PyContextToken_CheckExact (*C function*), 194
PyContextToken_Type (*C var*), 194
PyContextVar (*C type*), 194
PyContextVar_CheckExact (*C function*), 194
PyContextVar_Get (*C function*), 194
PyContextVar_New (*C function*), 194
PyContextVar_Reset (*C function*), 195
PyContextVar_Set (*C function*), 195
PyContextVar_Type (*C var*), 194
PyCoro_CheckExact (*C function*), 193
PyCoro_New (*C function*), 193
PyCoro_Type (*C var*), 193
PyCoroObject (*C type*), 193
PyDate_Check (*C function*), 196
PyDate_CheckExact (*C function*), 196
PyDate_FromDate (*C function*), 196
PyDate_FromTimestamp (*C function*), 198
PyDateTime_Check (*C function*), 196
PyDateTime_CheckExact (*C function*), 196
PyDateTime_Date (*C type*), 195
PyDateTime_DATE_GET_FOLD (*C function*), 197
PyDateTime_DATE_GET_HOUR (*C function*), 197
PyDateTime_DATE_GET_MICROSECOND (*C function*), 197
PyDateTime_DATE_GET_MINUTE (*C function*), 197
PyDateTime_DATE_GET_SECOND (*C function*), 197
PyDateTime_DATE_GET_TZINFO (*C function*), 197
PyDateTime_DateTime (*C type*), 195
PyDateTime_DateTimeType (*C var*), 195
PyDateTime_DateType (*C var*), 195
PyDateTime_Delta (*C type*), 195
PyDateTime_DELTA_GET_DAYS (*C function*), 198
PyDateTime_DELTA_GET_MICROSECONDS (*C function*), 198
PyDateTime_DELTA_GET_SECONDS (*C function*), 198
PyDateTime_DeltaType (*C var*), 195
PyDateTime_FromDateAndTime (*C function*), 196
PyDateTime_FromDateAndTimeAndFold (*C function*), 196
PyDateTime_FromTimestamp (*C function*), 198
PyDateTime_GET_DAY (*C function*), 197
PyDateTime_GET_MONTH (*C function*), 197
PyDateTime_GET_YEAR (*C function*), 197
PyDateTime_Time (*C type*), 195
PyDateTime_TIME_GET_FOLD (*C function*), 198
PyDateTime_TIME_GET_HOUR (*C function*), 197
PyDateTime_TIME_GET_MICROSECOND (*C function*), 198
PyDateTime_TIME_GET_MINUTE (*C function*), 197
PyDateTime_TIME_GET_SECOND (*C function*), 198
PyDateTime_TIME_GET_TZINFO (*C function*), 198
PyDateTime_TimeType (*C var*), 195
PyDateTime_TimeZone_UTC (*C var*), 195
PyDateTime_TZInfoType (*C var*), 195
PyDelta_Check (*C function*), 196
PyDelta_CheckExact (*C function*), 196
PyDelta_FromDSU (*C function*), 197
PyDescr_IsData (*C function*), 185
PyDescr_NewClassMethod (*C function*), 185
PyDescr_NewGetSet (*C function*), 185
PyDescr_NewMember (*C function*), 185
PyDescr_NewMethod (*C function*), 185
PyDescr_NewWrapper (*C function*), 185
PyDict_AddWatcher (*C function*), 165
PyDict_Check (*C function*), 161
PyDict_CheckExact (*C function*), 161
PyDict_Clear (*C function*), 162
PyDict_ClearWatcher (*C function*), 165
PyDict_Contains (*C function*), 162
PyDict_ContainsString (*C function*), 162
PyDict_Copy (*C function*), 162
PyDict_DelItem (*C function*), 162
PyDict_DelItemString (*C function*), 162
PyDict_GetItem (*C function*), 162
PyDict_GetItemRef (*C function*), 162
PyDict_GetItemString (*C function*), 163
PyDict_GetItemStringRef (*C function*), 163
PyDict_GetItemWithError (*C function*), 163
PyDict_Items (*C function*), 164
PyDict_Keys (*C function*), 164
PyDict_Merge (*C function*), 165
PyDict_MergeFromSeq2 (*C function*), 165
PyDict_New (*C function*), 161
PyDict_Next (*C function*), 164
PyDict_Pop (*C function*), 163
PyDict_PopString (*C function*), 163
PyDict_SetDefault (*C function*), 163
PyDict_SetDefaultRef (*C function*), 163
PyDict_SetItem (*C function*), 162
PyDict_SetItemString (*C function*), 162
PyDict_Size (*C function*), 164
PyDict_Type (*C var*), 161
PyDict_Unwatch (*C function*), 165
PyDict_Update (*C function*), 165
PyDict_Values (*C function*), 164
PyDict_Watch (*C function*), 165
PyDict_WatchCallback (*C type*), 165
PyDict_WatchEvent (*C type*), 165
PyDictObject (*C type*), 161
PyDictProxy_New (*C function*), 161
PyDoc_STR (*C macro*), 6
PyDoc_STRVAR (*C macro*), 6
PyErr_BadArgument (*C function*), 53
PyErr_BadInternalCall (*C function*), 54
PyErr_CheckSignals (*C function*), 59
PyErr_Clear (*C function*), 10, 12, 51
PyErr_DisplayException (*C function*), 52
PyErr_ExceptionMatches (*C function*), 12, 56
PyErr_Fetch (*C function*), 56
PyErr_Format (*C function*), 52

PyErr_FormatUnraisable (*C function*), 52
PyErr_FormatV (*C function*), 52
PyErr_GetExcInfo (*C function*), 58
PyErr_GetHandledException (*C function*), 57
PyErr_GetRaisedException (*C function*), 56
PyErr_GivenExceptionMatches (*C function*), 56
PyErr_NewException (*C function*), 60
PyErr_NewExceptionWithDoc (*C function*), 60
PyErr_NoMemory (*C function*), 53
PyErr_NormalizeException (*C function*), 57
PyErr_Occurred (*C function*), 10, 55
PyErr_Print (*C function*), 52
PyErr_PrintEx (*C function*), 51
PyErr_ResourceWarning (*C function*), 55
PyErr_Restore (*C function*), 57
PyErr_SetExcFromWindowsErr (*C function*), 53
PyErr_SetExcFromWindowsErrWithFilename (*C function*), 54
PyErr_SetExcFromWindowsErrWithFilenameObject (*C function*), 54
PyErr_SetExcFromWindowsErrWithFilenameObjects (*C function*), 54
PyErr_SetExcInfo (*C function*), 58
PyErr_SetFromErrno (*C function*), 53
PyErr_SetFromErrnoWithFilename (*C function*), 53
PyErr_SetFromErrnoWithFilenameObject (*C function*), 53
PyErr_SetFromErrnoWithFilenameObjects (*C function*), 53
PyErr_SetFromWindowsErr (*C function*), 53
PyErr_SetFromWindowsErrWithFilename (*C function*), 53
PyErr_SetHandledException (*C function*), 58
PyErr_SetImportError (*C function*), 54
PyErr_SetImportErrorSubclass (*C function*), 54
PyErr_SetInterrupt (*C function*), 59
PyErr_SetInterruptEx (*C function*), 59
PyErr_SetNone (*C function*), 53
PyErr_SetObject (*C function*), 52
PyErr_SetRaisedException (*C function*), 56
PyErr_SetString (*C function*), 10, 52
PyErr_SyntaxLocation (*C function*), 54
PyErr_SyntaxLocationEx (*C function*), 54
PyErr_SyntaxLocationObject (*C function*), 54
PyErr_WarnEx (*C function*), 55
PyErr_WarnExplicit (*C function*), 55
PyErr_WarnExplicitObject (*C function*), 55
PyErr_WarnFormat (*C function*), 55
PyErr_WriteUnraisable (*C function*), 52
PyEval_AcquireThread (*C function*), 217
PyEval_AcquireThread(), 213
PyEval_EvalCode (*C function*), 45
PyEval_EvalCodeEx (*C function*), 46
PyEval_EvalFrame (*C function*), 46
PyEval_EvalFrameEx (*C function*), 46
PyEval_GetBuiltins (*C function*), 86
PyEval_GetFrame (*C function*), 86
PyEval_GetFrameBuiltins (*C function*), 87
PyEval_GetFrameGlobals (*C function*), 87
PyEval_GetFrameLocals (*C function*), 87
PyEval_GetFuncDesc (*C function*), 87
PyEval_GetFuncName (*C function*), 87
PyEval_GetGlobals (*C function*), 86
PyEval_GetLocals (*C function*), 86
PyEval_InitThreads (*C function*), 213
PyEval_InitThreads(), 205
PyEval_MergeCompilerFlags (*C function*), 46
PyEval_ReleaseThread (*C function*), 217
PyEval_ReleaseThread(), 213
PyEval_RestoreThread (*C function*), 212, 213
PyEval_RestoreThread(), 213
PyEval_SaveThread (*C function*), 212, 213
PyEval_SaveThread(), 213
PyEval_SetProfile (*C function*), 222
PyEval_SetProfileAllThreads (*C function*), 222
PyEval_SetTrace (*C function*), 223
PyEval_SetTraceAllThreads (*C function*), 223
PyExc_ArithmeticError (*C var*), 63
PyExc_AssertionError (*C var*), 63
PyExc_AttributeError (*C var*), 63
PyExc_BaseException (*C var*), 63
PyExc_BlockingIOError (*C var*), 63
PyExc_BrokenPipeError (*C var*), 63
PyExc_BufferError (*C var*), 63
PyExc_BytesWarning (*C var*), 64
PyExc_ChildProcessError (*C var*), 63
PyExc_ConnectionAbortedError (*C var*), 63
PyExc_ConnectionError (*C var*), 63
PyExc_ConnectionRefusedError (*C var*), 63
PyExc_ConnectionResetError (*C var*), 63
PyExc_DeprecationWarning (*C var*), 64
PyExc_EnvironmentError (*C var*), 64
PyExc_EOFError (*C var*), 63
PyExc_Exception (*C var*), 63
PyExc_FileExistsError (*C var*), 63
PyExc_FileNotFoundError (*C var*), 63
PyExc_FloatingPointError (*C var*), 63
PyExc_FutureWarning (*C var*), 64
PyExc_GeneratorExit (*C var*), 63
PyExc_ImportError (*C var*), 63
PyExc_ImportWarning (*C var*), 64
PyExc_IndentationError (*C var*), 63
PyExc_IndexError (*C var*), 63
PyExc_InterruptedError (*C var*), 63
PyExc_IOError (*C var*), 64
PyExc_IsADirectoryError (*C var*), 63
PyExc_KeyboardInterrupt (*C var*), 63
PyExc_KeyError (*C var*), 63
PyExc_LookupError (*C var*), 63
PyExc_MemoryError (*C var*), 63
PyExc_ModuleNotFoundError (*C var*), 63
PyExc_NameError (*C var*), 63
PyExc_NotADirectoryError (*C var*), 63
PyExc_NotImplementedError (*C var*), 63
PyExc_OSError (*C var*), 63

PyExc_OverflowError (*C var*), 63
PyExc_PendingDeprecationWarning (*C var*), 64
PyExc_PermissionError (*C var*), 63
PyExc_ProcessLookupError (*C var*), 63
PyExc_PythonFinalizationError (*C var*), 63
PyExc_RecursionError (*C var*), 63
PyExc_ReferenceError (*C var*), 63
PyExc_ResourceWarning (*C var*), 64
PyExc_RuntimeError (*C var*), 63
PyExc_RuntimeWarning (*C var*), 64
PyExc_StopAsyncIteration (*C var*), 63
PyExc_StopIteration (*C var*), 63
PyExc_SyntaxError (*C var*), 63
PyExc_SyntaxWarning (*C var*), 64
PyExc_SystemError (*C var*), 63
PyExc_SystemExit (*C var*), 63
PyExc_TabError (*C var*), 63
PyExc_TimeoutError (*C var*), 63
PyExc_TypeError (*C var*), 63
PyExc_UnboundLocalError (*C var*), 63
PyExc_UnicodeDecodeError (*C var*), 63
PyExc_UnicodeEncodeError (*C var*), 63
PyExc_UnicodeError (*C var*), 63
PyExc_UnicodeTranslateError (*C var*), 63
PyExc_UnicodeWarning (*C var*), 64
PyExc_UserWarning (*C var*), 64
PyExc_ValueError (*C var*), 63
PyExc_Warning (*C var*), 64
PyExc_WindowsError (*C var*), 64
PyExc_ZeroDivisionError (*C var*), 63
PyException_GetArgs (*C function*), 60
PyException_GetCause (*C function*), 60
PyException_GetContext (*C function*), 60
PyException_GetTraceback (*C function*), 60
PyException_SetArgs (*C function*), 60
PyException_SetCause (*C function*), 60
PyException_SetContext (*C function*), 60
PyException_SetTraceback (*C function*), 60
PyFile_FromFd (*C function*), 175
PyFile_GetLine (*C function*), 175
PyFile_SetOpenCodeHook (*C function*), 175
PyFile_WriteObject (*C function*), 176
PyFile_WriteString (*C function*), 176
PyFloat_AS_DOUBLE (*C function*), 133
PyFloat_AsDouble (*C function*), 133
PyFloat_Check (*C function*), 133
PyFloat_CheckExact (*C function*), 133
PyFloat_FromDouble (*C function*), 133
PyFloat_FromString (*C function*), 133
PyFloat_GetInfo (*C function*), 133
PyFloat_GetMax (*C function*), 133
PyFloat_GetMin (*C function*), 133
PyFloat_Pack2 (*C function*), 134
PyFloat_Pack4 (*C function*), 134
PyFloat_Pack8 (*C function*), 134
PyFloat_Type (*C var*), 133
PyFloat_Unpack2 (*C function*), 134
PyFloat_Unpack4 (*C function*), 134
PyFloat_Unpack8 (*C function*), 134
PyFloatObject (*C type*), 133
PyFrame_Check (*C function*), 191
PyFrame_GetBack (*C function*), 191
PyFrame_GetBuiltins (*C function*), 191
PyFrame_GetCode (*C function*), 191
PyFrame_GetGenerator (*C function*), 191
PyFrame_GetGlobals (*C function*), 191
PyFrame_GetLasti (*C function*), 191
PyFrame_GetLineNumber (*C function*), 192
PyFrame_GetLocals (*C function*), 192
PyFrame_GetVar (*C function*), 191
PyFrame_GetVarString (*C function*), 191
PyFrame_Type (*C var*), 190
PyFrameObject (*C type*), 190
PyFrozenSet_Check (*C function*), 166
PyFrozenSet_CheckExact (*C function*), 167
PyFrozenSet_New (*C function*), 167
PyFrozenSet_Type (*C var*), 166
PyFunction_AddWatcher (*C function*), 169
PyFunction_Check (*C function*), 168
PyFunction_ClearWatcher (*C function*), 169
PyFunction_GetAnnotations (*C function*), 169
PyFunction_GetClosure (*C function*), 168
PyFunction_GetCode (*C function*), 168
PyFunction_GetDefaults (*C function*), 168
PyFunction_GetGlobals (*C function*), 168
PyFunction_GetModule (*C function*), 168
PyFunction_New (*C function*), 168
PyFunction_NewWithQualName (*C function*), 168
PyFunction_SetAnnotations (*C function*), 169
PyFunction_SetClosure (*C function*), 168
PyFunction_SetDefaults (*C function*), 168
PyFunction_SetVectorcall (*C function*), 168
PyFunction_Type (*C var*), 168
PyFunction_WatchCallback (*C type*), 169
PyFunction_WatchEvent (*C type*), 169
PyFunctionObject (*C type*), 168
PyGC_Collect (*C function*), 310
PyGC_Disable (*C function*), 310
PyGC_Enable (*C function*), 310
PyGC_IsEnabled (*C function*), 310
PyGen_Check (*C function*), 193
PyGen_CheckExact (*C function*), 193
PyGen_New (*C function*), 193
PyGen_NewWithQualName (*C function*), 193
PyGen_Type (*C var*), 193
PyGenObject (*C type*), 193
PyGetSetDef (*C type*), 272
PyGetSetDef.closure (*C member*), 272
PyGetSetDef.doc (*C member*), 272
PyGetSetDef.get (*C member*), 272
PyGetSetDef.name (*C member*), 272
PyGetSetDef.set (*C member*), 272
PyGILState_Check (*C function*), 214
PyGILState_Ensure (*C function*), 214
PyGILState_GetThisThreadState (*C function*), 214

PyGILState_Release (*C function*), 214
PyHASH_BITS (*C macro*), 85
PyHash_FuncDef (*C type*), 85
PyHash_FuncDef.hash_bits (*C member*), 85
PyHash_FuncDef.name (*C member*), 85
PyHash_FuncDef.seed_bits (*C member*), 85
PyHash_GetFuncDef (*C function*), 85
PyHASH_IMAG (*C macro*), 85
PyHASH_INF (*C macro*), 85
PyHASH_MODULUS (*C macro*), 85
PyHASH_MULTIPLIER (*C macro*), 85
PyImport_AddModule (*C function*), 71
PyImport_AddModuleObject (*C function*), 71
PyImport_AddModuleRef (*C function*), 71
PyImport_AppendInittab (*C function*), 73
PyImport_ExecCodeModule (*C function*), 71
PyImport_ExecCodeModuleEx (*C function*), 72
PyImport_ExecCodeModuleObject (*C function*), 72
PyImport_ExecCodeModuleWithPathnames (*C function*), 72
PyImport_ExtendInittab (*C function*), 73
PyImport_FrozenModules (*C var*), 73
PyImport_GetImporter (*C function*), 72
PyImport_GetMagicNumber (*C function*), 72
PyImport_GetMagicTag (*C function*), 72
PyImport_GetModule (*C function*), 72
PyImport_GetModuleDict (*C function*), 72
PyImport_Import (*C function*), 71
PyImport_ImportFrozenModule (*C function*), 73
PyImport_ImportFrozenModuleObject (*C function*), 73
PyImport_ImportModule (*C function*), 70
PyImport_ImportModuleEx (*C function*), 70
PyImport_ImportModuleLevel (*C function*), 71
PyImport_ImportModuleLevelObject (*C function*), 70
PyImport_ImportModuleNoBlock (*C function*), 70
PyImport_ReloadModule (*C function*), 71
PyIndex_Check (*C function*), 107
PyInstanceMethod_Check (*C function*), 170
PyInstanceMethod_Function (*C function*), 170
PyInstanceMethod_GET_FUNCTION (*C function*), 170
PyInstanceMethod_New (*C function*), 170
PyInstanceMethod_Type (*C var*), 170
PyInterpreterConfig (*C type*), 218
PyInterpreterConfig_DEFAULT_GIL (*C macro*), 218
PyInterpreterConfig_OWN_GIL (*C macro*), 218
PyInterpreterConfig_SHARED_GIL (*C macro*), 218
PyInterpreterConfig.allow_daemon_threads (*C member*), 218
PyInterpreterConfig.allow_exec (*C member*), 218
PyInterpreterConfig.allow_fork (*C member*), 218
PyInterpreterConfig.allow_threads (*C member*), 218
PyInterpreterConfig.check_multi_interp_extensions (*C member*), 218
PyInterpreterConfig.gil (*C member*), 218
PyInterpreterConfig.use_main_obmalloc (*C member*), 218
PyInterpreterState (*C type*), 213
PyInterpreterState_Clear (*C function*), 215
PyInterpreterState_Delete (*C function*), 215
PyInterpreterState_Get (*C function*), 216
PyInterpreterState_GetDict (*C function*), 216
PyInterpreterState_GetID (*C function*), 216
PyInterpreterState_Head (*C function*), 224
PyInterpreterState_Main (*C function*), 224
PyInterpreterState_New (*C function*), 215
PyInterpreterState_Next (*C function*), 224
PyInterpreterState_ThreadHead (*C function*), 224
PyIter_Check (*C function*), 111
PyIter_Next (*C function*), 111
PyIter_Send (*C function*), 112
PyList_Append (*C function*), 160
PyList_AsTuple (*C function*), 161
PyList_Check (*C function*), 159
PyList_CheckExact (*C function*), 159
PyList_Clear (*C function*), 161
PyList_Extend (*C function*), 161
PyList_GET_ITEM (*C function*), 160
PyList_GET_SIZE (*C function*), 160
PyList_GetItem (*C function*), 9, 160
PyList_GetItemRef (*C function*), 160
PyList_GetSlice (*C function*), 160
PyList_Insert (*C function*), 160
PyList_New (*C function*), 159
PyList_Reverse (*C function*), 161
PyList_SET_ITEM (*C function*), 160
PyList_SetItem (*C function*), 8, 160
PyList_SetSlice (*C function*), 161
PyList_Size (*C function*), 160
PyList_Sort (*C function*), 161
PyList_Type (*C var*), 159
PyListObject (*C type*), 159
PyLong_AS_LONG (*C function*), 127
PyLong_AsDouble (*C function*), 128
PyLong_AsInt (*C function*), 127
PyLong_AsLong (*C function*), 127
PyLong_AsLongAndOverflow (*C function*), 127
PyLong_AsLongLong (*C function*), 127
PyLong_AsLongLongAndOverflow (*C function*), 127
PyLong_AsNativeBytes (*C function*), 129
PyLong_AsSize_t (*C function*), 128
PyLong_AsSsize_t (*C function*), 128
PyLong_AsUnsignedLong (*C function*), 128
PyLong_AsUnsignedLongLong (*C function*), 128
PyLong_AsUnsignedLongLongMask (*C function*), 128
PyLong_AsUnsignedLongMask (*C function*), 128

PyLong_AsVoidPtr (*C function*), 129
PyLong_Check (*C function*), 125
PyLong_CheckExact (*C function*), 125
PyLong_FromDouble (*C function*), 126
PyLong_FromLong (*C function*), 125
PyLong_FromLongLong (*C function*), 126
PyLong_FromNativeBytes (*C function*), 126
PyLong_FromSize_t (*C function*), 126
PyLong_FromSsize_t (*C function*), 126
PyLong_FromString (*C function*), 126
PyLong_FromUnicodeObject (*C function*), 126
PyLong_FromUnsignedLong (*C function*), 125
PyLong_FromUnsignedLongLong (*C function*), 126
PyLong_FromUnsignedNativeBytes (*C function*), 126
PyLong_FromVoidPtr (*C function*), 126
PyLong_GetInfo (*C function*), 131
PyLong_Type (*C var*), 125
PyLongObject (*C type*), 125
PyMapping_Check (*C function*), 109
PyMapping_DelItem (*C function*), 110
PyMapping_DelItemString (*C function*), 110
PyMapping_GetItemString (*C function*), 110
PyMapping_GetOptionalItem (*C function*), 110
PyMapping_GetOptionalItemString (*C function*), 110
PyMapping_HasKey (*C function*), 110
PyMapping_HasKeyString (*C function*), 110
PyMapping_HasKeyStringWithError (*C function*), 110
PyMapping_HasKeyWithError (*C function*), 110
PyMapping_Items (*C function*), 111
PyMapping_Keys (*C function*), 111
PyMapping_Length (*C function*), 110
PyMapping_SetItemString (*C function*), 110
PyMapping_Size (*C function*), 110
PyMapping_Values (*C function*), 111
PyMappingMethods (*C type*), 301
PyMappingMethods.mp_ass_subscript (*C member*), 301
PyMappingMethods.mp_length (*C member*), 301
PyMappingMethods.mp_subscript (*C member*), 301
PyMarshal_ReadLastObjectFromFile (*C function*), 74
PyMarshal_ReadLongFromFile (*C function*), 74
PyMarshal_ReadObjectFromFile (*C function*), 74
PyMarshal_ReadObjectFromString (*C function*), 74
PyMarshal_ReadShortFromFile (*C function*), 74
PyMarshal_WriteLongToFile (*C function*), 74
PyMarshal_WriteObjectToFile (*C function*), 74
PyMarshal_WriteObjectToString (*C function*), 74
PyMem_Calloc (*C function*), 253
PyMem_Del (*C function*), 254
PYMEM_DOMAIN_MEM (*C macro*), 256
PYMEM_DOMAIN_OBJ (*C macro*), 257
PYMEM_DOMAIN_RAW (*C macro*), 256
PyMem_Free (*C function*), 254
PyMem_GetAllocator (*C function*), 257
PyMem_Malloc (*C function*), 253
PyMem_New (*C macro*), 254
PyMem_RawCalloc (*C function*), 253
PyMem_RawFree (*C function*), 253
PyMem_RawMalloc (*C function*), 253
PyMem_RawRealloc (*C function*), 253
PyMem_Realloc (*C function*), 254
PyMem_Resize (*C macro*), 254
PyMem_SetAllocator (*C function*), 257
PyMem_SetupDebugHooks (*C function*), 257
PyMemAllocatorDomain (*C type*), 256
PyMemAllocatorEx (*C type*), 256
PyMember_GetOne (*C function*), 269
PyMember_SetOne (*C function*), 269
PyMemberDef (*C type*), 269
PyMemberDef.doc (*C member*), 269
PyMemberDef.flags (*C member*), 269
PyMemberDef.name (*C member*), 269
PyMemberDef.offset (*C member*), 269
PyMemberDef.type (*C member*), 269
PyMemoryView_Check (*C function*), 187
PyMemoryView_FromBuffer (*C function*), 187
PyMemoryView_FromMemory (*C function*), 187
PyMemoryView_FromObject (*C function*), 187
PyMemoryView_GET_BASE (*C function*), 187
PyMemoryView_GET_BUFFER (*C function*), 187
PyMemoryView_GetContiguous (*C function*), 187
PyMethod_Check (*C function*), 170
PyMethod_Function (*C function*), 170
PyMethod_GET_FUNCTION (*C function*), 170
PyMethod_GET_SELF (*C function*), 170
PyMethod_New (*C function*), 170
PyMethod_Self (*C function*), 170
PyMethod_Type (*C var*), 170
PyMethodDef (*C type*), 266
PyMethodDef.ml_doc (*C member*), 267
PyMethodDef.ml_flags (*C member*), 266
PyMethodDef.ml_meth (*C member*), 266
PyMethodDef.ml_name (*C member*), 266
PyMODINIT_FUNC (*C macro*), 4
PyModule_Add (*C function*), 182
PyModule_AddFunctions (*C function*), 181
PyModule_AddIntConstant (*C function*), 183
PyModule_AddIntMacro (*C macro*), 183
PyModule_AddObject (*C function*), 182
PyModule_AddObjectRef (*C function*), 181
PyModule_AddStringConstant (*C function*), 183
PyModule_AddStringMacro (*C macro*), 183
PyModule_AddType (*C function*), 183
PyModule_Check (*C function*), 176
PyModule_CheckExact (*C function*), 176
PyModule_Create (*C function*), 178
PyModule_Create2 (*C function*), 178
PyModule_ExecDef (*C function*), 181
PyModule_FromDefAndSpec (*C function*), 180
PyModule_FromDefAndSpec2 (*C function*), 181

PyModule_GetDef (*C function*), 177
PyModule_GetDict (*C function*), 176
PyModule_GetFilename (*C function*), 177
PyModule_GetFilenameObject (*C function*), 177
PyModule_GetName (*C function*), 176
PyModule_GetNameObject (*C function*), 176
PyModule_GetState (*C function*), 176
PyModule_New (*C function*), 176
PyModule_NewObject (*C function*), 176
PyModule_SetDocString (*C function*), 181
PyModule_Type (*C var*), 176
PyModuleDef (*C type*), 177
PyModuleDef_Init (*C function*), 179
PyModuleDef_Slot (*C type*), 179
PyModuleDef_Slot.slot (*C member*), 179
PyModuleDef_Slot.value (*C member*), 179
PyModuleDef.m_base (*C member*), 177
PyModuleDef.m_clear (*C member*), 178
PyModuleDef.m_doc (*C member*), 177
PyModuleDef.m_free (*C member*), 178
PyModuleDef.m_methods (*C member*), 177
PyModuleDef.m_name (*C member*), 177
PyModuleDef.m_size (*C member*), 177
PyModuleDef.m_slots (*C member*), 177
PyModuleDef.m_slots.m_reload (*C member*), 178
PyModuleDef.m_traverse (*C member*), 178
PyMonitoring_EnterScope (*C function*), 318
PyMonitoring_ExitScope (*C function*), 319
PyMonitoring_FireBranchEvent (*C function*), 318
PyMonitoring_FireCallEvent (*C function*), 317
PyMonitoring_FireCRaiseEvent (*C function*), 318
PyMonitoring_FireCReturnEvent (*C function*), 318
PyMonitoring_FireExceptionHandledEvent (*C function*), 318
PyMonitoring_FireJumpEvent (*C function*), 317
PyMonitoring_FireLineEvent (*C function*), 317
PyMonitoring_FirePyResumeEvent (*C function*), 317
PyMonitoring_FirePyReturnEvent (*C function*), 317
PyMonitoring_FirePyStartEvent (*C function*), 317
PyMonitoring_FirePyThrowEvent (*C function*), 318
PyMonitoring_FirePyUnwindEvent (*C function*), 318
PyMonitoring_FirePyYieldEvent (*C function*), 317
PyMonitoring_FireRaiseEvent (*C function*), 318
PyMonitoring_FireReraiseEvent (*C function*), 318
PyMonitoring_FireStopIterationEvent (*C function*), 318
PyMonitoringState (*C type*), 317
PyMutex (*C type*), 226
PyMutex_Lock (*C function*), 226
PyMutex_Unlock (*C function*), 226
PyNumber_Absolute (*C function*), 105
PyNumber_Add (*C function*), 105
PyNumber_And (*C function*), 106
PyNumber_AsSsize_t (*C function*), 107
PyNumber_Check (*C function*), 104
PyNumber_Divmod (*C function*), 105
PyNumber_Float (*C function*), 107
PyNumber_FloorDivide (*C function*), 105
PyNumber_Index (*C function*), 107
PyNumber_InPlaceAdd (*C function*), 106
PyNumber_InPlaceAnd (*C function*), 107
PyNumber_InPlaceFloorDivide (*C function*), 106
PyNumber_InPlaceLshift (*C function*), 107
PyNumber_InPlaceMatrixMultiply (*C function*), 106
PyNumber_InPlaceMultiply (*C function*), 106
PyNumber_InPlaceOr (*C function*), 107
PyNumber_InPlacePower (*C function*), 106
PyNumber_InPlaceRemainder (*C function*), 106
PyNumber_InPlaceRshift (*C function*), 107
PyNumber_InPlaceSubtract (*C function*), 106
PyNumber_InPlaceTrueDivide (*C function*), 106
PyNumber_InPlaceXor (*C function*), 107
PyNumber_Invert (*C function*), 105
PyNumber_Long (*C function*), 107
PyNumber_Lshift (*C function*), 106
PyNumber_MatrixMultiply (*C function*), 105
PyNumber_Multiply (*C function*), 105
PyNumber_Negative (*C function*), 105
PyNumber_Or (*C function*), 106
PyNumber_Positive (*C function*), 105
PyNumber_Power (*C function*), 105
PyNumber_Remainder (*C function*), 105
PyNumber_Rshift (*C function*), 106
PyNumber_Subtract (*C function*), 105
PyNumber_ToBase (*C function*), 107
PyNumber_TrueDivide (*C function*), 105
PyNumber_Xor (*C function*), 106
PyNumberMethods (*C type*), 298
PyNumberMethods.nb_absolute (*C member*), 300
PyNumberMethods.nb_add (*C member*), 299
PyNumberMethods.nb_and (*C member*), 300
PyNumberMethods.nb_bool (*C member*), 300
PyNumberMethods.nb_divmod (*C member*), 300
PyNumberMethods.nb_float (*C member*), 300
PyNumberMethods.nb_floor_divide (*C member*), 300
PyNumberMethods.nb_index (*C member*), 300
PyNumberMethods.nb_inplace_add (*C member*), 300
PyNumberMethods.nb_inplace_and (*C member*), 300
PyNumberMethods.nb_inplace_floor_divide (*C member*), 300
PyNumberMethods.nb_inplace_lshift (*C member*), 300
PyNumberMethods.nb_inplace_matrix_multiply (*C member*), 300

PyNumberMethods.nb_inplace_multiply (*C member*), 300
PyNumberMethods.nb_inplace_or (*C member*), 300
PyNumberMethods.nb_inplace_power (*C member*), 300
PyNumberMethods.nb_inplace_remainder (*C member*), 300
PyNumberMethods.nb_inplace_rshift (*C member*), 300
PyNumberMethods.nb_inplace_subtract (*C member*), 300
PyNumberMethods.nb_inplace_true_divide (*C member*), 300
PyNumberMethods.nb_inplace_xor (*C member*), 300
PyNumberMethods.nb_int (*C member*), 300
PyNumberMethods.nb_invert (*C member*), 300
PyNumberMethods.nb_lshift (*C member*), 300
PyNumberMethods.nb_matrix_multiply (*C member*), 300
PyNumberMethods.nb_multiply (*C member*), 299
PyNumberMethods.nb_negative (*C member*), 300
PyNumberMethods.nb_or (*C member*), 300
PyNumberMethods.nb_positive (*C member*), 300
PyNumberMethods.nb_power (*C member*), 300
PyNumberMethods.nb_remainder (*C member*), 299
PyNumberMethods.nb_reserved (*C member*), 300
PyNumberMethods.nb_rshift (*C member*), 300
PyNumberMethods.nb_subtract (*C member*), 299
PyNumberMethods.nb_true_divide (*C member*), 300
PyNumberMethods.nb_xor (*C member*), 300
PyObject (*C type*), 264
PyObject_ASCII (*C function*), 97
PyObject_AsFileDescriptor (*C function*), 175
PyObject_Bytes (*C function*), 97
PyObject_Call (*C function*), 102
PyObject_CallFunction (*C function*), 103
PyObject_CallFunctionObjArgs (*C function*), 103
PyObject_CallMethod (*C function*), 103
PyObject_CallMethodNoArgs (*C function*), 103
PyObject_CallMethodObjArgs (*C function*), 103
PyObject_CallMethodOneArg (*C function*), 104
PyObject_CallNoArgs (*C function*), 102
PyObject_CallObject (*C function*), 103
PyObject_Calloc (*C function*), 255
PyObject_CallOneArg (*C function*), 103
PyObject_CheckBuffer (*C function*), 117
PyObject_ClearManagedDict (*C function*), 100
PyObject_ClearWeakRefs (*C function*), 188
PyObject_CopyData (*C function*), 118
PyObject_Del (*C function*), 263
PyObject_DelAttr (*C function*), 96
PyObject_DelAttrString (*C function*), 96
PyObject_DelItem (*C function*), 99
PyObject_Dir (*C function*), 99
PyObject_Format (*C function*), 97
PyObject_Free (*C function*), 255
PyObject_GC_Del (*C function*), 309
PyObject_GC_IsFinalized (*C function*), 309
PyObject_GC_IsTracked (*C function*), 309
PyObject_GC_New (*C macro*), 308
PyObject_GC_NewVar (*C macro*), 308
PyObject_GC_Resize (*C macro*), 309
PyObject_GC_Track (*C function*), 309
PyObject_GC_UnTrack (*C function*), 309
PyObject_GenericGetAttr (*C function*), 96
PyObject_GenericGetDict (*C function*), 96
PyObject_GenericHash (*C function*), 86
PyObject_GenericSetAttr (*C function*), 96
PyObject_GenericSetDict (*C function*), 97
PyObject_GetAIter (*C function*), 99
PyObject_GetArenaAllocator (*C function*), 259
PyObject_GetAttr (*C function*), 95
PyObject_GetAttrString (*C function*), 95
PyObject_GetBuffer (*C function*), 117
PyObject_GetItem (*C function*), 99
PyObject_GetItemData (*C function*), 100
PyObject_GetIter (*C function*), 99
PyObject_GetOptionalAttr (*C function*), 95
PyObject_GetOptionalAttrString (*C function*), 96
PyObject_GetTypeData (*C function*), 99
PyObject_HasAttr (*C function*), 95
PyObject_HasAttrString (*C function*), 95
PyObject_HasAttrStringWithError (*C function*), 95
PyObject_HasAttrWithError (*C function*), 95
PyObject_Hash (*C function*), 98
PyObject_HashNotImplemented (*C function*), 98
PyObject_HEAD (*C macro*), 264
PyObject_HEAD_INIT (*C macro*), 265
PyObject_Init (*C function*), 263
PyObject_InitVar (*C function*), 263
PyObject_IS_GC (*C function*), 309
PyObject_IsInstance (*C function*), 98
PyObject_IsSubclass (*C function*), 98
PyObject_IsTrue (*C function*), 98
PyObject_Length (*C function*), 99
PyObject_LengthHint (*C function*), 99
PyObject_Malloc (*C function*), 255
PyObject_New (*C macro*), 263
PyObject_NewVar (*C macro*), 263
PyObject_Not (*C function*), 98
PyObject_Print (*C function*), 94
PyObject_Realloc (*C function*), 255
PyObject_Repr (*C function*), 97
PyObject_RichCompare (*C function*), 97
PyObject_RichCompareBool (*C function*), 97
PyObject_SetArenaAllocator (*C function*), 259
PyObject_SetAttr (*C function*), 96
PyObject_SetAttrString (*C function*), 96
PyObject_SetItem (*C function*), 99
PyObject_Size (*C function*), 99
PyObject_Str (*C function*), 97

PyObject_Type (*C function*), 98
PyObject_TypeCheck (*C function*), 98
PyObject_VAR_HEAD (*C macro*), 264
PyObject_Vectorcall (*C function*), 104
PyObject_VectorcallDict (*C function*), 104
PyObject_VectorcallMethod (*C function*), 104
PyObject_VisitManagedDict (*C function*), 100
PyObjectArenaAllocator (*C type*), 259
PyObject.ob_refcnt (*C member*), 278
PyObject.ob_type (*C member*), 278
PyOS_AfterFork (*C function*), 66
PyOS_AfterFork_Child (*C function*), 66
PyOS_AfterFork_Parent (*C function*), 65
PyOS_BeforeFork (*C function*), 65
PyOS_CheckStack (*C function*), 66
PyOS_double_to_string (*C function*), 84
PyOS_FSPath (*C function*), 65
PyOS_getsig (*C function*), 66
PyOS_InputHook (*C var*), 44
PyOS_ReadlineFunctionPointer (*C var*), 44
PyOS_setsig (*C function*), 66
PyOS_sighandler_t (*C type*), 66
PyOS_snprintf (*C function*), 83
PyOS_stricmp (*C function*), 85
PyOS_string_to_double (*C function*), 84
PyOS_strnicmp (*C function*), 85
PyOS_strtol (*C function*), 84
PyOS_strtoul (*C function*), 83
PyOS_vsnprintf (*C function*), 83
PyPreConfig (*C type*), 232
PyPreConfig_InitIsolatedConfig (*C function*), 232
PyPreConfig_InitPythonConfig (*C function*), 232
PyPreConfig.allocator (*C member*), 232
PyPreConfig.coerce_c_locale (*C member*), 232
PyPreConfig.coerce_c_locale_warn (*C member*), 233
PyPreConfig.configure_locale (*C member*), 232
PyPreConfig.dev_mode (*C member*), 233
PyPreConfig.isolated (*C member*), 233
PyPreConfig.legacy_windows_fs_encoding (*C member*), 233
PyPreConfig.parse_argv (*C member*), 233
PyPreConfig.use_environment (*C member*), 233
PyPreConfig.utf8_mode (*C member*), 233
PyProperty_Type (*C var*), 185
PyRefTracer (*C type*), 223
PyRefTracer_CREATE (*C var*), 223
PyRefTracer_DESTROY (*C var*), 223
PyRefTracer_GetTracer (*C function*), 223
PyRefTracer_SetTracer (*C function*), 223
PyRun_AnyFile (*C function*), 43
PyRun_AnyFileEx (*C function*), 43
PyRun_AnyFileExFlags (*C function*), 43
PyRun_AnyFileFlags (*C function*), 43
PyRun_File (*C function*), 45
PyRun_FileEx (*C function*), 45
PyRun_FileExFlags (*C function*), 45
PyRun_FileFlags (*C function*), 45
PyRun_InteractiveLoop (*C function*), 44
PyRun_InteractiveLoopFlags (*C function*), 44
PyRun_InteractiveOne (*C function*), 44
PyRun_InteractiveOneFlags (*C function*), 44
PyRun_SimpleFile (*C function*), 44
PyRun_SimpleFileEx (*C function*), 44
PyRun_SimpleFileExFlags (*C function*), 44
PyRun_SimpleString (*C function*), 43
PyRun_SimpleStringFlags (*C function*), 43
PyRun_String (*C function*), 45
PyRun_StringFlags (*C function*), 45
PySendResult (*C type*), 112
PySeqIter_Check (*C function*), 184
PySeqIter_New (*C function*), 184
PySeqIter_Type (*C var*), 184
PySequence_Check (*C function*), 108
PySequence_Concat (*C function*), 108
PySequence_Contains (*C function*), 109
PySequence_Count (*C function*), 108
PySequence_DelItem (*C function*), 108
PySequence_DelSlice (*C function*), 108
PySequence_Fast (*C function*), 109
PySequence_Fast_GET_ITEM (*C function*), 109
PySequence_Fast_GET_SIZE (*C function*), 109
PySequence_Fast_ITEMS (*C function*), 109
PySequence_GetItem (*C function*), 9, 108
PySequence_GetSlice (*C function*), 108
PySequence_Index (*C function*), 109
PySequence_InPlaceConcat (*C function*), 108
PySequence_InPlaceRepeat (*C function*), 108
PySequence_ITEM (*C function*), 109
PySequence_Length (*C function*), 108
PySequence_List (*C function*), 109
PySequence_Repeat (*C function*), 108
PySequence_SetItem (*C function*), 108
PySequence_SetSlice (*C function*), 108
PySequence_Size (*C function*), 108
PySequence_Tuple (*C function*), 109
PySequenceMethods (*C type*), 301
PySequenceMethods.sq_ass_item (*C member*), 301
PySequenceMethods.sq_concat (*C member*), 301
PySequenceMethods.sq_contains (*C member*), 301
PySequenceMethods.sq_inplace_concat (*C member*), 301
PySequenceMethods.sq_inplace_repeat (*C member*), 302
PySequenceMethods.sq_item (*C member*), 301
PySequenceMethods.sq_length (*C member*), 301
PySequenceMethods.sq_repeat (*C member*), 301
PySet_Add (*C function*), 167
PySet_Check (*C function*), 166
PySet_CheckExact (*C function*), 166
PySet_Clear (*C function*), 167
PySet_Contains (*C function*), 167
PySet_Discard (*C function*), 167

PySet_GET_SIZE (*C function*), 167
PySet_New (*C function*), 167
PySet_Pop (*C function*), 167
PySet_Size (*C function*), 167
PySet_Type (*C var*), 166
PySetObject (*C type*), 166
PySignal_SetWakeupFd (*C function*), 59
PySlice_AdjustIndices (*C function*), 186
PySlice_Check (*C function*), 185
PySlice_GetIndices (*C function*), 185
PySlice_GetIndicesEx (*C function*), 185
PySlice_New (*C function*), 185
PySlice_Type (*C var*), 185
PySlice_Unpack (*C function*), 186
PyState_AddModule (*C function*), 184
PyState_FindModule (*C function*), 184
PyState_RemoveModule (*C function*), 184
PyStatus (*C type*), 230
PyStatus_Error (*C function*), 231
PyStatus_Exception (*C function*), 231
PyStatus_Exit (*C function*), 231
PyStatus_IsError (*C function*), 231
PyStatus_IsExit (*C function*), 231
PyStatus_NoMemory (*C function*), 231
PyStatus_Ok (*C function*), 231
PyStatus.err_msg (*C member*), 231
PyStatus.exitcode (*C member*), 231
PyStatus.func (*C member*), 231
PyStructSequence_Desc (*C type*), 158
PyStructSequence_Desc.doc (*C member*), 158
PyStructSequence_Desc.fields (*C member*), 158
PyStructSequence_Desc.n_in_sequence (*C member*), 158
PyStructSequence_Desc.name (*C member*), 158
PyStructSequence_Field (*C type*), 158
PyStructSequence_Field.doc (*C member*), 158
PyStructSequence_Field.name (*C member*), 158
PyStructSequence_GET_ITEM (*C function*), 159
PyStructSequence_GetItem (*C function*), 159
PyStructSequence_InitType (*C function*), 158
PyStructSequence_InitType2 (*C function*), 158
PyStructSequence_New (*C function*), 159
PyStructSequence_NewType (*C function*), 158
PyStructSequence_SET_ITEM (*C function*), 159
PyStructSequence_SetItem (*C function*), 159
PyStructSequence_UnnamedField (*C var*), 159
PySys_AddAuditHook (*C function*), 69
PySys_Audit (*C function*), 68
PySys_AuditTuple (*C function*), 69
PySys_FormatStderr (*C function*), 68
PySys_FormatStdout (*C function*), 68
PySys_GetObject (*C function*), 68
PySys_GetXOptions (*C function*), 68
PySys_ResetWarnOptions (*C function*), 68
PySys_SetArgv (*C function*), 210
PySys_SetArgvEx (*C function*), 210
PySys_SetObject (*C function*), 68
PySys_WriteStderr (*C function*), 68
PySys_WriteStdout (*C function*), 68
Python 3000, **333**
Python Enhancement Proposals
    PEP 1, 333
    PEP 7, 3, 6
    PEP 238, 46, 326
    PEP 278, 336
    PEP 302, 330
    PEP 343, 324
    PEP 353, 10
    PEP 362, 322, 332
    PEP 383, 147, 148
    PEP 387, 15
    PEP 393, 139
    PEP 411, 333
    PEP 420, 331, 333
    PEP 432, 249
    PEP 442, 298
    PEP 443, 327
    PEP 451, 179
    PEP 456, 86
    PEP 483, 327
    PEP 484, 321, 327, 336
    PEP 489, 180, 218
    PEP 492, 322, 324
    PEP 498, 326
    PEP 519, 333
    PEP 523, 192, 217
    PEP 525, 322
    PEP 526, 321, 336
    PEP 528, 204, 240
    PEP 529, 148, 204
    PEP 538, 247
    PEP 539, 224
    PEP 540, 247
    PEP 552, 237
    PEP 554, 220
    PEP 578, 69
    PEP 585, 327
    PEP 587, 229
    PEP 590, 100
    PEP 623, 139
    PEP 0626#out-of-process-debuggers-and-profilers, 172
    PEP 634, 287, 288
    PEP 667, 86, 192
    PEP 0683, 47, 48, 328
    PEP 703, 326, 328
    PEP 3116, 336
    PEP 3119, 98
    PEP 3121, 177
    PEP 3147, 72
    PEP 3151, 64
    PEP 3155, 334
PYTHON_CPU_COUNT, 240
PYTHON_GIL, 328
PYTHON_PERF_JIT_SUPPORT, 244
PYTHON_PRESITE, 243

PYTHONCOERCECLOCALE, 247
PYTHONDEBUG, 203, 242
PYTHONDEVMODE, 238
PYTHONDONTWRITEBYTECODE, 203, 245
PYTHONDUMPREFS, 238
PYTHONEXECUTABLE, 242
PYTHONFAULTHANDLER, 238
PYTHONHASHSEED, 203, 239
PYTHONHOME, 12, 203, 210, 211, 239
Pythonic, **333**
PYTHONINSPECT, 203, 239
PYTHONINTMAXSTRDIGITS, 240
PYTHONIOENCODING, 243
PYTHONLEGACYWINDOWSFSENCODING, 204, 233
PYTHONLEGACYWINDOWSSTDIO, 204, 240
PYTHONMALLOC, 252, 256, 258, 259
PYTHONMALLOCSTATS, 240, 252
PYTHONNODEBUGRANGES, 237
PYTHONNOUSERSITE, 204, 244
PYTHONOPTIMIZE, 204, 241
PYTHONPATH, 12, 203, 241
PYTHONPLATLIBDIR, 240
PYTHONPROFILEIMPORTTIME, 239
PYTHONPYCACHEPREFIX, 242
PYTHONSAFEPATH, 236
PYTHONTRACEMALLOC, 244
PYTHONUNBUFFERED, 205, 237
PYTHONUTF8, 233, 247
PYTHONVERBOSE, 205, 244
PYTHONWARNINGS, 245
PyThread_create_key (*C function*), 226
PyThread_delete_key (*C function*), 226
PyThread_delete_key_value (*C function*), 226
PyThread_get_key_value (*C function*), 226
PyThread_ReInitTLS (*C function*), 226
PyThread_set_key_value (*C function*), 226
PyThread_tss_alloc (*C function*), 225
PyThread_tss_create (*C function*), 225
PyThread_tss_delete (*C function*), 225
PyThread_tss_free (*C function*), 225
PyThread_tss_get (*C function*), 225
PyThread_tss_is_created (*C function*), 225
PyThread_tss_set (*C function*), 225
PyThreadState (*C type*), 211, 213
PyThreadState_Clear (*C function*), 215
PyThreadState_Delete (*C function*), 215
PyThreadState_DeleteCurrent (*C function*), 215
PyThreadState_EnterTracing (*C function*), 216
PyThreadState_Get (*C function*), 213
PyThreadState_GetDict (*C function*), 217
PyThreadState_GetFrame (*C function*), 215
PyThreadState_GetID (*C function*), 216
PyThreadState_GetInterpreter (*C function*), 216
PyThreadState_GetUnchecked (*C function*), 213
PyThreadState_LeaveTracing (*C function*), 216
PyThreadState_New (*C function*), 215
PyThreadState_Next (*C function*), 224
PyThreadState_SetAsyncExc (*C function*), 217
PyThreadState_Swap (*C function*), 214
PyThreadState.interp (*C member*), 213
PyTime_AsSecondsDouble (*C function*), 90
PyTime_Check (*C function*), 196
PyTime_CheckExact (*C function*), 196
PyTime_FromTime (*C function*), 196
PyTime_FromTimeAndFold (*C function*), 196
PyTime_MAX (*C var*), 89
PyTime_MIN (*C var*), 89
PyTime_Monotonic (*C function*), 89
PyTime_MonotonicRaw (*C function*), 90
PyTime_PerfCounter (*C function*), 89
PyTime_PerfCounterRaw (*C function*), 90
PyTime_t (*C type*), 89
PyTime_Time (*C function*), 89
PyTime_TimeRaw (*C function*), 90
PyTimeZone_FromOffset (*C function*), 197
PyTimeZone_FromOffsetAndName (*C function*), 197
PyTrace_C_CALL (*C var*), 222
PyTrace_C_EXCEPTION (*C var*), 222
PyTrace_C_RETURN (*C var*), 222
PyTrace_CALL (*C var*), 222
PyTrace_EXCEPTION (*C var*), 222
PyTrace_LINE (*C var*), 222
PyTrace_OPCODE (*C var*), 222
PyTrace_RETURN (*C var*), 222
PyTraceMalloc_Track (*C function*), 260
PyTraceMalloc_Untrack (*C function*), 260
PyTuple_Check (*C function*), 157
PyTuple_CheckExact (*C function*), 157
PyTuple_GET_ITEM (*C function*), 157
PyTuple_GET_SIZE (*C function*), 157
PyTuple_GetItem (*C function*), 157
PyTuple_GetSlice (*C function*), 157
PyTuple_New (*C function*), 157
PyTuple_Pack (*C function*), 157
PyTuple_SET_ITEM (*C function*), 157
PyTuple_SetItem (*C function*), 8, 157
PyTuple_Size (*C function*), 157
PyTuple_Type (*C var*), 156
PyTupleObject (*C type*), 156
PyType_AddWatcher (*C function*), 120
PyType_Check (*C function*), 119
PyType_CheckExact (*C function*), 119
PyType_ClearCache (*C function*), 119
PyType_ClearWatcher (*C function*), 120
PyType_FromMetaclass (*C function*), 122
PyType_FromModuleAndSpec (*C function*), 123
PyType_FromSpec (*C function*), 123
PyType_FromSpecWithBases (*C function*), 123
PyType_GenericAlloc (*C function*), 121
PyType_GenericNew (*C function*), 121
PyType_GetDict (*C function*), 120
PyType_GetFlags (*C function*), 119
PyType_GetFullyQualifiedName (*C function*), 121
PyType_GetModule (*C function*), 122
PyType_GetModuleByDef (*C function*), 122
PyType_GetModuleName (*C function*), 121

PyType_GetModuleState (*C function*), 122
PyType_GetName (*C function*), 121
PyType_GetQualName (*C function*), 121
PyType_GetSlot (*C function*), 121
PyType_GetTypeDataSize (*C function*), 99
PyType_HasFeature (*C function*), 120
PyType_IS_GC (*C function*), 120
PyType_IsSubtype (*C function*), 120
PyType_Modified (*C function*), 120
PyType_Ready (*C function*), 121
PyType_Slot (*C type*), 124
PyType_Slot.pfunc (*C member*), 125
PyType_Slot.slot (*C member*), 124
PyType_Spec (*C type*), 123
PyType_Spec.basicsize (*C member*), 123
PyType_Spec.flags (*C member*), 124
PyType_Spec.itemsize (*C member*), 124
PyType_Spec.name (*C member*), 123
PyType_Spec.slots (*C member*), 124
PyType_Type (*C var*), 119
PyType_Watch (*C function*), 120
PyType_WatchCallback (*C type*), 120
PyTypeObject (*C type*), 119
PyTypeObject.tp_alloc (*C member*), 295
PyTypeObject.tp_as_async (*C member*), 281
PyTypeObject.tp_as_buffer (*C member*), 284
PyTypeObject.tp_as_mapping (*C member*), 282
PyTypeObject.tp_as_number (*C member*), 282
PyTypeObject.tp_as_sequence (*C member*), 282
PyTypeObject.tp_base (*C member*), 292
PyTypeObject.tp_bases (*C member*), 296
PyTypeObject.tp_basicsize (*C member*), 279
PyTypeObject.tp_cache (*C member*), 296
PyTypeObject.tp_call (*C member*), 283
PyTypeObject.tp_clear (*C member*), 289
PyTypeObject.tp_dealloc (*C member*), 280
PyTypeObject.tp_del (*C member*), 297
PyTypeObject.tp_descr_get (*C member*), 293
PyTypeObject.tp_descr_set (*C member*), 293
PyTypeObject.tp_dict (*C member*), 293
PyTypeObject.tp_dictoffset (*C member*), 294
PyTypeObject.tp_doc (*C member*), 288
PyTypeObject.tp_finalize (*C member*), 297
PyTypeObject.tp_flags (*C member*), 284
PyTypeObject.tp_free (*C member*), 295
PyTypeObject.tp_getattr (*C member*), 281
PyTypeObject.tp_getattro (*C member*), 283
PyTypeObject.tp_getset (*C member*), 292
PyTypeObject.tp_hash (*C member*), 282
PyTypeObject.tp_init (*C member*), 294
PyTypeObject.tp_is_gc (*C member*), 296
PyTypeObject.tp_itemsize (*C member*), 279
PyTypeObject.tp_iter (*C member*), 292
PyTypeObject.tp_iternext (*C member*), 292
PyTypeObject.tp_members (*C member*), 292
PyTypeObject.tp_methods (*C member*), 292
PyTypeObject.tp_mro (*C member*), 296
PyTypeObject.tp_name (*C member*), 279
PyTypeObject.tp_new (*C member*), 295
PyTypeObject.tp_repr (*C member*), 281
PyTypeObject.tp_richcompare (*C member*), 290
PyTypeObject.tp_setattr (*C member*), 281
PyTypeObject.tp_setattro (*C member*), 283
PyTypeObject.tp_str (*C member*), 283
PyTypeObject.tp_subclasses (*C member*), 296
PyTypeObject.tp_traverse (*C member*), 288
PyTypeObject.tp_vectorcall (*C member*), 298
PyTypeObject.tp_vectorcall_offset (*C member*), 281
PyTypeObject.tp_version_tag (*C member*), 297
PyTypeObject.tp_watched (*C member*), 298
PyTypeObject.tp_weaklist (*C member*), 296
PyTypeObject.tp_weaklistoffset (*C member*), 291
PyTZInfo_Check (*C function*), 196
PyTZInfo_CheckExact (*C function*), 196
PyUnicode_1BYTE_DATA (*C function*), 140
PyUnicode_1BYTE_KIND (*C macro*), 140
PyUnicode_2BYTE_DATA (*C function*), 140
PyUnicode_2BYTE_KIND (*C macro*), 140
PyUnicode_4BYTE_DATA (*C function*), 140
PyUnicode_4BYTE_KIND (*C macro*), 140
PyUnicode_AsASCIIString (*C function*), 153
PyUnicode_AsCharmapString (*C function*), 153
PyUnicode_AsEncodedString (*C function*), 150
PyUnicode_AsLatin1String (*C function*), 153
PyUnicode_AsMBCSString (*C function*), 154
PyUnicode_AsRawUnicodeEscapeString (*C function*), 153
PyUnicode_AsUCS4 (*C function*), 147
PyUnicode_AsUCS4Copy (*C function*), 147
PyUnicode_AsUnicodeEscapeString (*C function*), 152
PyUnicode_AsUTF8 (*C function*), 151
PyUnicode_AsUTF8AndSize (*C function*), 150
PyUnicode_AsUTF8String (*C function*), 150
PyUnicode_AsUTF16String (*C function*), 152
PyUnicode_AsUTF32String (*C function*), 151
PyUnicode_AsWideChar (*C function*), 149
PyUnicode_AsWideCharString (*C function*), 149
PyUnicode_Check (*C function*), 140
PyUnicode_CheckExact (*C function*), 140
PyUnicode_Compare (*C function*), 155
PyUnicode_CompareWithASCIIString (*C function*), 155
PyUnicode_Concat (*C function*), 154
PyUnicode_Contains (*C function*), 156
PyUnicode_CopyCharacters (*C function*), 146
PyUnicode_Count (*C function*), 155
PyUnicode_DATA (*C function*), 140
PyUnicode_Decode (*C function*), 150
PyUnicode_DecodeASCII (*C function*), 153
PyUnicode_DecodeCharmap (*C function*), 153
PyUnicode_DecodeFSDefault (*C function*), 148
PyUnicode_DecodeFSDefaultAndSize (*C function*), 148

PyUnicode_DecodeLatin1 (*C function*), 153
PyUnicode_DecodeLocale (*C function*), 147
PyUnicode_DecodeLocaleAndSize (*C function*), 147
PyUnicode_DecodeMBCS (*C function*), 154
PyUnicode_DecodeMBCSStateful (*C function*), 154
PyUnicode_DecodeRawUnicodeEscape (*C function*), 153
PyUnicode_DecodeUnicodeEscape (*C function*), 152
PyUnicode_DecodeUTF7 (*C function*), 152
PyUnicode_DecodeUTF7Stateful (*C function*), 152
PyUnicode_DecodeUTF8 (*C function*), 150
PyUnicode_DecodeUTF8Stateful (*C function*), 150
PyUnicode_DecodeUTF16 (*C function*), 151
PyUnicode_DecodeUTF16Stateful (*C function*), 152
PyUnicode_DecodeUTF32 (*C function*), 151
PyUnicode_DecodeUTF32Stateful (*C function*), 151
PyUnicode_EncodeCodePage (*C function*), 154
PyUnicode_EncodeFSDefault (*C function*), 149
PyUnicode_EncodeLocale (*C function*), 147
PyUnicode_EqualToUTF8 (*C function*), 155
PyUnicode_EqualToUTF8AndSize (*C function*), 155
PyUnicode_Fill (*C function*), 146
PyUnicode_Find (*C function*), 155
PyUnicode_FindChar (*C function*), 155
PyUnicode_Format (*C function*), 156
PyUnicode_FromEncodedObject (*C function*), 146
PyUnicode_FromFormat (*C function*), 143
PyUnicode_FromFormatV (*C function*), 146
PyUnicode_FromKindAndData (*C function*), 143
PyUnicode_FromObject (*C function*), 146
PyUnicode_FromString (*C function*), 143
PyUnicode_FromStringAndSize (*C function*), 143
PyUnicode_FromWideChar (*C function*), 149
PyUnicode_FSConverter (*C function*), 148
PyUnicode_FSDecoder (*C function*), 148
PyUnicode_GET_LENGTH (*C function*), 140
PyUnicode_GetLength (*C function*), 146
PyUnicode_InternFromString (*C function*), 156
PyUnicode_InternInPlace (*C function*), 156
PyUnicode_IsIdentifier (*C function*), 141
PyUnicode_Join (*C function*), 155
PyUnicode_KIND (*C function*), 140
PyUnicode_MAX_CHAR_VALUE (*C function*), 141
PyUnicode_New (*C function*), 142
PyUnicode_READ (*C function*), 141
PyUnicode_READ_CHAR (*C function*), 141
PyUnicode_ReadChar (*C function*), 146
PyUnicode_READY (*C function*), 140
PyUnicode_Replace (*C function*), 155
PyUnicode_RichCompare (*C function*), 156
PyUnicode_Split (*C function*), 154
PyUnicode_Splitlines (*C function*), 155
PyUnicode_Substring (*C function*), 147
PyUnicode_Tailmatch (*C function*), 155
PyUnicode_Translate (*C function*), 154
PyUnicode_Type (*C var*), 140
PyUnicode_WRITE (*C function*), 141
PyUnicode_WriteChar (*C function*), 146
PyUnicodeDecodeError_Create (*C function*), 61
PyUnicodeDecodeError_GetEncoding (*C function*), 61
PyUnicodeDecodeError_GetEnd (*C function*), 61
PyUnicodeDecodeError_GetObject (*C function*), 61
PyUnicodeDecodeError_GetReason (*C function*), 62
PyUnicodeDecodeError_GetStart (*C function*), 61
PyUnicodeDecodeError_SetEnd (*C function*), 61
PyUnicodeDecodeError_SetReason (*C function*), 62
PyUnicodeDecodeError_SetStart (*C function*), 61
PyUnicodeEncodeError_GetEncoding (*C function*), 61
PyUnicodeEncodeError_GetEnd (*C function*), 61
PyUnicodeEncodeError_GetObject (*C function*), 61
PyUnicodeEncodeError_GetReason (*C function*), 62
PyUnicodeEncodeError_GetStart (*C function*), 61
PyUnicodeEncodeError_SetEnd (*C function*), 61
PyUnicodeEncodeError_SetReason (*C function*), 62
PyUnicodeEncodeError_SetStart (*C function*), 61
PyUnicodeObject (*C type*), 139
PyUnicodeTranslateError_GetEnd (*C function*), 61
PyUnicodeTranslateError_GetObject (*C function*), 61
PyUnicodeTranslateError_GetReason (*C function*), 62
PyUnicodeTranslateError_GetStart (*C function*), 61
PyUnicodeTranslateError_SetEnd (*C function*), 61
PyUnicodeTranslateError_SetReason (*C function*), 62
PyUnicodeTranslateError_SetStart (*C function*), 61
PyUnstable, 15
PyUnstable_Code_GetExtra (*C function*), 174
PyUnstable_Code_GetFirstFree (*C function*), 171
PyUnstable_Code_New (*C function*), 172
PyUnstable_Code_NewWithPosOnlyArgs (*C function*), 172
PyUnstable_Code_SetExtra (*C function*), 174
PyUnstable_Eval_RequestCodeExtraIndex (*C function*), 174
PyUnstable_Exc_PrepReraiseStar (*C function*), 60
PyUnstable_GC_VisitObjects (*C function*), 310
PyUnstable_InterpreterFrame_GetCode (*C function*), 192

PyUnstable_InterpreterFrame_GetLasti (*C function*), 192
PyUnstable_InterpreterFrame_GetLine (*C function*), 192
PyUnstable_Long_CompactValue (*C function*), 132
PyUnstable_Long_IsCompact (*C function*), 131
PyUnstable_Module_SetGIL (*C function*), 183
PyUnstable_Object_ClearWeakRefsNoCallbacks (*C function*), 188
PyUnstable_Object_GC_NewWithExtraData (*C function*), 308
PyUnstable_PerfMapState_Fini (*C function*), 91
PyUnstable_PerfMapState_Init (*C function*), 90
PyUnstable_Type_AssignVersionTag (*C function*), 122
PyUnstable_WritePerfMapEntry (*C function*), 90
PyVarObject (*C type*), 264
PyVarObject_HEAD_INIT (*C macro*), 265
PyVarObject.ob_size (*C member*), 279
PyVectorcall_Call (*C function*), 102
PyVectorcall_Function (*C function*), 102
PyVectorcall_NARGS (*C function*), 101
PyWeakref_Check (*C function*), 187
PyWeakref_CheckProxy (*C function*), 187
PyWeakref_CheckRef (*C function*), 187
PyWeakref_GET_OBJECT (*C function*), 188
PyWeakref_GetObject (*C function*), 188
PyWeakref_GetRef (*C function*), 188
PyWeakref_NewProxy (*C function*), 188
PyWeakref_NewRef (*C function*), 187
PyWideStringList (*C type*), 230
PyWideStringList_Append (*C function*), 230
PyWideStringList_Insert (*C function*), 230
PyWideStringList.items (*C member*), 230
PyWideStringList.length (*C member*), 230
PyWrapper_New (*C function*), 185

Q

qualified name, **334**

R

READ_RESTRICTED (*C macro*), 270
READONLY (*C macro*), 270
realloc (*C function*), 251
reference count, **334**
regular package, **334**
releasebufferproc (*C type*), 305
REPL, **334**
repr
    funkcja wbudowana, 97, 281
reprfunc (*C type*), 304
RESTRICTED (*C macro*), 270
richcmpfunc (*C type*), 304

S

sendfunc (*C type*), 305
sequence, **334**
    obiekt, 136
set
    obiekt, 166
set comprehension, **334**
set_all(), 9
setattrfunc (*C type*), 304
setattrofunc (*C type*), 304
setswitchinterval (*in module sys*), 211
setter (*C type*), 272
SIGINT (*C macro*), 59
signal
    moduł, 59
single dispatch, **334**
SIZE_MAX (*C macro*), 128
slice, **335**
słownik, **325**
    obiekt, 161
soft deprecated, **335**
special
    method, 335
special method, **335**
ssizeargfunc (*C type*), 305
ssizeobjargproc (*C type*), 305
static type checker, **335**
staticmethod
    funkcja wbudowana, 268
stderr (*in module sys*), 219, 220
stdin (*in module sys*), 219, 220
stdout (*in module sys*), 219, 220
strerror (*C function*), 53
string
    PyObject_Str (*C function*), 97
strong reference, **335**
structmember.h, 272
sum_list(), 9
sum_sequence(), 10, 11
sys
    moduł, 12, 205, 219, 220
SystemError (*built-in exception*), 176, 177

Ś

ścieżka
    moduł wyszukiwanie, 12, 205, 209

T

T_BOOL (*C macro*), 272
T_BYTE (*C macro*), 272
T_CHAR (*C macro*), 272
T_DOUBLE (*C macro*), 272
T_FLOAT (*C macro*), 272
T_INT (*C macro*), 272
T_LONG (*C macro*), 272
T_LONGLONG (*C macro*), 272
T_NONE (*C macro*), 272
T_OBJECT (*C macro*), 272
T_OBJECT_EX (*C macro*), 272
T_PYSSIZET (*C macro*), 272
T_SHORT (*C macro*), 272
T_STRING (*C macro*), 272

T_STRING_INPLACE (*C macro*), 272
T_UBYTE (*C macro*), 272
T_UINT (*C macro*), 272
T_ULONG (*C macro*), 272
T_ULONGULONG (*C macro*), 272
T_USHORT (*C macro*), 272
ternaryfunc (*C type*), 305
text encoding, **335**
text file, **335**
traverseproc (*C type*), 309
triple-quoted string, **335**
tuple
    funkcja wbudowana, 109, 161
    obiekt, 156
type, **335**
    funkcja wbudowana, 98
    obiekt, 7, 119
type alias, **335**
type hint, **336**

U

ULONG_MAX (*C macro*), 128
unaryfunc (*C type*), 305
universal newlines, **336**
USE_STACKCHECK (*C macro*), 66

V

variable annotation, **336**
vectorcallfunc (*C type*), 101
version (*in module sys*), 209, 210
virtual environment, **336**
virtual machine, **336**
visitproc (*C type*), 309

W

WRITE_RESTRICTED (*C macro*), 270
wyszukiwanie
    ścieżka, moduł, 12, 205, 209
wywołanie zwrotne, **323**

Z

Zen of Python, **336**
zmienna środowiskowa
    __PYVENV_LAUNCHER__, 236, 242
    PATH, 12
    PYTHON_CPU_COUNT, 240
    PYTHON_GIL, 328
    PYTHON_PERF_JIT_SUPPORT, 244
    PYTHON_PRESITE, 243
    PYTHONCOERCECLOCALE, 247
    PYTHONDEBUG, 203, 242
    PYTHONDEVMODE, 238
    PYTHONDONTWRITEBYTECODE, 203, 245
    PYTHONDUMPREFS, 238
    PYTHONEXECUTABLE, 242
    PYTHONFAULTHANDLER, 238
    PYTHONHASHSEED, 203, 239
    PYTHONHOME, 12, 203, 210, 211, 239
    PYTHONINSPECT, 203, 239
    PYTHONINTMAXSTRDIGITS, 240
    PYTHONIOENCODING, 243
    PYTHONLEGACYWINDOWSFSENCODING, 204, 233
    PYTHONLEGACYWINDOWSSTDIO, 204, 240
    PYTHONMALLOC, 252, 256, 258, 259
    PYTHONMALLOCSTATS, 240, 252
    PYTHONNODEBUGRANGES, 237
    PYTHONNOUSERSITE, 204, 244
    PYTHONOPTIMIZE, 204, 241
    PYTHONPATH, 12, 203, 241
    PYTHONPLATLIBDIR, 240
    PYTHONPROFILEIMPORTTIME, 239
    PYTHONPYCACHEPREFIX, 242
    PYTHONSAFEPATH, 236
    PYTHONTRACEMALLOC, 244
    PYTHONUNBUFFERED, 205, 237
    PYTHONUTF8, 233, 247
    PYTHONVERBOSE, 205, 244
    PYTHONWARNINGS, 245