
The Python/C API

Sürüm 3.9.20

**Guido van Rossum
and the Python development team**

Eylül 08, 2024

**Python Software Foundation
Email: docs@python.org**

1	Introduction	3
1.1	Coding standards	3
1.2	Include Files	3
1.3	Useful macros	4
1.4	Objects, Types and Reference Counts	6
1.4.1	Reference Counts	6
1.4.2	Types	9
1.5	Exceptions	9
1.6	Embedding Python	11
1.7	Debugging Builds	12
2	Stable Application Binary Interface	13
3	The Very High Level Layer	15
4	Reference Counting	21
5	Exception Handling	23
5.1	Printing and clearing	23
5.2	Raising exceptions	24
5.3	Issuing warnings	26
5.4	Querying the error indicator	27
5.5	Signal Handling	29
5.6	Exception Classes	29
5.7	Exception Objects	29
5.8	Unicode Exception Objects	30
5.9	Recursion Control	31
5.10	Standard Exceptions	32
5.11	Standard Warning Categories	33
6	Utilities	35
6.1	Operating System Utilities	35
6.2	System Functions	37
6.3	Process Control	39
6.4	Importing Modules	40
6.5	Data marshallng support	43
6.6	Parsing arguments and building values	44
6.6.1	Parsing arguments	44
6.6.2	Building values	49
6.7	String conversion and formatting	51
6.8	Reflection	53

6.9	Codec registry and support functions	53
6.9.1	Codec lookup API	54
6.9.2	Registry API for Unicode encoding error handlers	54
7	Abstract Objects Layer	57
7.1	Object Protocol	57
7.2	Call Protocol	61
7.2.1	The <i>tp_call</i> Protocol	61
7.2.2	The Vectorcall Protocol	61
7.2.3	Object Calling API	63
7.2.4	Call Support API	65
7.3	Number Protocol	65
7.4	Sequence Protocol	68
7.5	Mapping Protocol	70
7.6	Iterator Protocol	71
7.7	Buffer Protocol	71
7.7.1	Buffer structure	72
7.7.2	Buffer request types	74
7.7.3	Complex arrays	76
7.7.4	Buffer-related functions	77
7.8	Old Buffer Protocol	78
8	Concrete Objects Layer	79
8.1	Fundamental Objects	79
8.1.1	Type Objects	79
8.1.2	The <code>None</code> Object	82
8.2	Numeric Objects	83
8.2.1	Integer Objects	83
8.2.2	Boolean Objects	86
8.2.3	Floating Point Objects	86
8.2.4	Complex Number Objects	87
8.3	Sequence Objects	88
8.3.1	Bytes Objects	88
8.3.2	Byte Array Objects	90
8.3.3	Unicode Objects and Codecs	91
8.3.4	Tuple Objects	109
8.3.5	Struct Sequence Objects	110
8.3.6	List Objects	112
8.4	Container Objects	113
8.4.1	Dictionary Objects	113
8.4.2	Set Objects	115
8.5	Function Objects	117
8.5.1	Function Objects	117
8.5.2	Instance Method Objects	118
8.5.3	Method Objects	118
8.5.4	Cell Objects	119
8.5.5	Code Objects	119
8.6	Other Objects	120
8.6.1	File Objects	120
8.6.2	Module Objects	121
8.6.3	Iterator Objects	127
8.6.4	Descriptor Objects	127
8.6.5	Slice Objects	128
8.6.6	Ellipsis Object	129
8.6.7	MemoryView objects	129
8.6.8	Weak Reference Objects	130
8.6.9	Capsules	131
8.6.10	Generator Objects	132

8.6.11	Coroutine Objects	133
8.6.12	Context Variables Objects	133
8.6.13	DateTime Objects	134
8.6.14	Objects for Type Hinting	137
9	Initialization, Finalization, and Threads	139
9.1	Before Python Initialization	139
9.2	Global configuration variables	140
9.3	Initializing and finalizing the interpreter	142
9.4	Process-wide parameters	143
9.5	Thread State and the Global Interpreter Lock	146
9.5.1	Releasing the GIL from extension code	146
9.5.2	Non-Python created threads	147
9.5.3	Cautions about fork()	147
9.5.4	High-level API	148
9.5.5	Low-level API	150
9.6	Sub-interpreter support	152
9.6.1	Bugs and caveats	153
9.7	Asynchronous Notifications	154
9.8	Profiling and Tracing	154
9.9	Advanced Debugger Support	156
9.10	Thread Local Storage Support	156
9.10.1	Thread Specific Storage (TSS) API	156
9.10.2	Thread Local Storage (TLS) API	157
10	Python Initialization Configuration	159
10.1	PyWideStringList	160
10.2	PyStatus	161
10.3	PyPreConfig	162
10.4	Preinitialization with PyPreConfig	163
10.5	PyConfig	164
10.6	Initialization with PyConfig	168
10.7	Isolated Configuration	170
10.8	Python Configuration	170
10.9	Path Configuration	171
10.10	Py_RunMain()	172
10.11	Py_GetArgcArgv()	172
10.12	Multi-Phase Initialization Private Provisional API	172
11	Memory Management	175
11.1	Overview	175
11.2	Raw Memory Interface	176
11.3	Memory Interface	177
11.4	Object allocators	178
11.5	Default Memory Allocators	179
11.6	Customize Memory Allocators	179
11.7	The pymalloc allocator	181
11.7.1	Customize pymalloc Arena Allocator	181
11.8	tracemalloc C API	181
11.9	Examples	182
12	Object Implementation Support	183
12.1	Allocating Objects on the Heap	183
12.2	Common Object Structures	184
12.2.1	Base object types and macros	184
12.2.2	Implementing functions and methods	185
12.2.3	Accessing attributes of extension types	187
12.3	Type Objects	189
12.3.1	Quick Reference	189

12.3.2	PyTypeObject Definition	194
12.3.3	PyObject Slots	195
12.3.4	PyVarObject Slots	196
12.3.5	PyTypeObject Slots	196
12.3.6	Heap Types	212
12.4	Number Object Structures	212
12.5	Mapping Object Structures	214
12.6	Sequence Object Structures	214
12.7	Buffer Object Structures	215
12.8	Async Object Structures	216
12.9	Slot Type typedefs	217
12.10	Examples	218
12.11	Supporting Cyclic Garbage Collection	220
13	API and ABI Versioning	223
A	Sözlük	225
B	Dokümanlar hakkında	239
B.1	Python Dokümantasyonuna Katkıda Bulunanlar	239
C	Tarihçe ve Lisans	241
C.1	Yazılımın tarihçesi	241
C.2	Python'a erişmek veya başka bir şekilde kullanmak için şartlar ve koşullar	242
C.2.1	PYTHON İÇİN PSF LİSANS ANLAŞMASI 3.9.20	242
C.2.2	PYTHON 2.0 İÇİN BEOPEN.COM LİSANS SÖZLEŞMESİ	243
C.2.3	PYTHON 1.6.1 İÇİN CNRI LİSANS ANLAŞMASI	244
C.2.4	0.9.0 ARASI 1.2 PYTHON İÇİN CWI LİSANS SÖZLEŞMESİ	245
C.2.5	PYTHON 3.9.20 BELGELERİNDEKİ KOD İÇİN SIFIR MADDE BSD LİSANSI	245
C.3	Tüzel Yazılımlar için Lisanslar ve Onaylar	245
C.3.1	Mersenne Twister'ı	246
C.3.2	Soketler	246
C.3.3	Asenkron soket hizmetleri	247
C.3.4	Çerez yönetimi	247
C.3.5	Çalıştırma izleme	248
C.3.6	UUencode ve UUdecode fonksiyonları	248
C.3.7	XML Uzaktan Yordam Çağrıları	249
C.3.8	test_epoll	249
C.3.9	kqueue seçin	250
C.3.10	SipHash24	250
C.3.11	strtod ve dtoa	251
C.3.12	OpenSSL	251
C.3.13	expat	253
C.3.14	libffi	254
C.3.15	zlib	254
C.3.16	cfuhash	255
C.3.17	libmpdec	256
C.3.18	W3C C14N test paketi	256
D	Telif Hakkı	259
	Dizin	261

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.

The Application Programmer's Interface to Python gives C and C++ programmers access to the Python interpreter at a variety of levels. The API is equally usable from C++, but for brevity it is generally referred to as the Python/C API. There are two fundamentally different reasons for using the Python/C API. The first reason is to write *extension modules* for specific purposes; these are C modules that extend the Python interpreter. This is probably the most common use. The second reason is to use Python as a component in a larger application; this technique is generally referred to as *embedding* Python in an application.

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.

Many API functions are useful independent of whether you're embedding or extending Python; moreover, most applications that embed Python will need to provide a custom extension as well, so it's probably a good idea to become familiar with writing an extension before attempting to embed Python in a real application.

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 Include Files

All function, type and macro definitions needed to use the Python/C API are included in your code by the following line:

```
#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).

Not: Since Python may define some pre-processor definitions which affect the standard headers on some systems, you *must* include `Python.h` before any standard headers are included.

It is recommended to always define `PY_SSIZE_T_CLEAN` before including `Python.h`. See [Parsing arguments and building values](#) for a description of this macro.

All user visible names defined by `Python.h` (except those defined by the included standard headers) have one of the prefixes `Py` or `_Py`. Names beginning with `_Py` are for internal use by the Python implementation and should not be used by extension writers. Structure member names do not have a reserved prefix.

Not: 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.

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.

3.7 sürümüyle geldi.

Py_ABS(x)

Return the absolute value of `x`.

3.3 sürümüyle geldi.

Py_MIN(x, y)

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

3.3 sürümüyle geldi.

Py_MAX(x, y)

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

3.3 sürümüyle geldi.

Py_STRINGIFY (x)

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

3.4 sürümüyle geldi.

Py_MEMBER_SIZE (type, member)

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

3.6 sürümüyle geldi.

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_GETENV (s)

Like `getenv(s)`, but returns NULL if `-E` was passed on the command line (i.e. if `Py_IgnoreEnvironmentFlag` is set).

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; }`.

3.4 sürümüyle geldi.

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);
```

3.8 sürümünde değişti: MSVC support was added.

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 sqlite_row_methods[] = {
    {"keys", (PyCFunction)sqlite_row_keys, METH_NOARGS,
     PyDoc_STR("Returns the keys of the row.")},
    {NULL, NULL}
};
```

1.4 Objects, Types and Reference Counts

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.

All Python objects (even Python integers) have a *type* and a *reference count*. An object's type determines what kind of object it is (e.g., an integer, a list, or a user-defined function; there are many more as explained in types). For each of the well-known types there is a macro to check whether an object is of that type; for instance, `PyList_Check(a)` is true if (and only if) the object pointed to by *a* is a Python list.

1.4.1 Reference Counts

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 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 an object's reference count becomes zero, the object is deallocated. If it contains references to other objects, their reference count is decremented. Those other objects may be deallocated in turn, if this decrement makes their reference count become zero, 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 increment an object's reference count by one, and `Py_DECREF()` to decrement it by one. The `Py_DECREF()` macro is considerably more complex than the `Py_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 decrementing the reference counts 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 increment an object's 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 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 incrementing its reference count. Some other operation might conceivably remove the object from the list, decrementing its reference count 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 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.

Reference Count Details

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 decref’ing 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.

Conversely, when a calling function passes in a reference to an object, there are two possibilities: the function *steals* a reference to the object, or it does not. *Stealing a reference* means that when you pass a reference to a function, that function assumes that it now owns that reference, and you are not responsible for it any longer.

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 reference counts is much saner, since you don’t have to increment a reference count so you can give a 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;
```

(continues on next page)

(önceki sayfadan devam)

```

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;
    n = PySequence_Length(sequence);
    if (n < 0)
        return -1; /* Has no length */
    for (i = 0; i < n; i++) {

```

(continues on next page)

(önceki sayfadan devam)

```

    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 Types

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.

`Py_ssize_t`

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 Exceptions

The Python programmer only needs to deal with exceptions if specific error handling is required; unhandled exceptions are automatically propagated to the caller, then to the caller's caller, and so on, until they reach the top-level interpreter, where they are reported to the user accompanied by a stack traceback.

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.

Note that starting with Python 1.5, the preferred, thread-safe way to access the exception state from Python code is to call the function `sys.exc_info()`, which returns the per-thread exception state for Python code. Also, the semantics of both ways to access the exception state have changed so that a function which catches an exception will save and restore its thread's exception state so as to preserve the exception state of its caller. This prevents common bugs in exception handling code caused by an innocent-looking function overwriting the exception being handled; it also reduces the often unwanted lifetime extension for objects that are referenced by the stack frames in the traceback.

As a general principle, a function that calls another function to perform some task should check whether the called function raised an exception, and if so, pass the exception state on to its caller. It should discard any object references that it owns, and return an error indicator, but it should *not* set another exception — that would overwrite the exception that was just raised, and lose important information about the exact cause of the error.

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
```

Here is the corresponding C code, in all its glory:

```
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;

    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 */
```

(continues on next page)

(önceki sayfadan devam)

```

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 Embedding Python

The one important task that only embedders (as opposed to extension writers) of the Python interpreter have to worry about is the initialization, and possibly the finalization, of the Python interpreter. Most functionality of the interpreter can only be used after the interpreter has been initialized.

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, it must be set explicitly with a call to `PySys_SetArgvEx(argc, argv, updatepath)` after the call to `Py_Initialize()`.

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`).

For instance, if the Python executable is found in `/usr/local/bin/python`, it will assume that the libraries are in `/usr/local/lib/pythonX.Y`. (In fact, this particular path is also the “fallback” location, used when no executable file named `python` is found along `PATH`.) The user can override this behavior by setting the environment variable `PYTHONHOME`, or insert additional directories in front of the standard path by setting `PYTHONPATH`.

The embedding application can steer the search by calling `Py_SetProgramName(file)` *before* calling `Py_Initialize()`. 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 Debugging Builds

Python can be built with several macros to enable extra checks of the interpreter and extension modules. These checks tend to add a large amount of overhead to the runtime so they are not enabled by default.

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.

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, the following extra checks are performed:

- Extra checks are added to the object allocator.
- Extra checks are added to the parser and compiler.
- Downcasts from wide types to narrow types are checked for loss of information.
- A number of assertions are added to the dictionary and set implementations. In addition, the set object acquires a `test_c_api()` method.
- Sanity checks of the input arguments are added to frame creation.
- The storage for ints is initialized with a known invalid pattern to catch reference to uninitialized digits.
- Low-level tracing and extra exception checking are added to the runtime virtual machine.
- Extra checks are added to the memory arena implementation.
- Extra debugging is added to the thread module.

There may be additional checks not mentioned here.

Defining `Py_TRACE_REFS` enables reference tracing. 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.) Implied by `Py_DEBUG`.

Please refer to `Misc/SpecialBuilds.txt` in the Python source distribution for more detailed information.

Stable Application Binary Interface

Traditionally, the C API of Python will change with every release. Most changes will be source-compatible, typically by only adding API, rather than changing existing API or removing API (although some interfaces do get removed after being deprecated first).

Unfortunately, the API compatibility does not extend to binary compatibility (the ABI). The reason is primarily the evolution of struct definitions, where addition of a new field, or changing the type of a field, might not break the API, but can break the ABI. As a consequence, extension modules need to be recompiled for every Python release (although an exception is possible on Unix when none of the affected interfaces are used). In addition, on Windows, extension modules link with a specific pythonXY.dll and need to be recompiled to link with a newer one.

Since Python 3.2, a subset of the API has been declared to guarantee a stable ABI. Extension modules wishing to use this API (called “limited API”) need to define `PY_LIMITED_API`. A number of interpreter details then become hidden from the extension module; in return, a module is built that works on any 3.x version ($x \geq 2$) without recompilation.

In some cases, the stable ABI needs to be extended with new functions. Extension modules wishing to use these new APIs need to set `PY_LIMITED_API` to the `PY_VERSION_HEX` value (see [API and ABI Versioning](#)) of the minimum Python version they want to support (e.g. `0x03030000` for Python 3.3). Such modules will work on all subsequent Python releases, but fail to load (because of missing symbols) on the older releases.

As of Python 3.2, the set of functions available to the limited API is documented in [PEP 384](#). In the C API documentation, API elements that are not part of the limited API are marked as “Not part of the limited API.”

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 **Py_Main** (int *argc*, wchar_t ***argv*)

The main program for the standard interpreter. This is made available for programs which embed Python. The *argc* and *argv* parameters should be prepared exactly as those which are passed to a C program's `main()` function (converted to `wchar_t` according to the user's locale). It is important to note that the argument list may be modified (but the contents of the strings pointed to by the argument list are not). 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 parameter 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.

int **Py_BytesMain** (int *argc*, char ***argv*)

Similar to `Py_Main()` but *argv* is an array of bytes strings.

3.8 sürümüyle geldi.

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 `Py_InspectFlag` is not set.

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 the filesystem encoding (*sys.getfilesystemencoding()*). If *closeit* is true, the file is closed before *PyRun_SimpleFileExFlags* returns.

Not: 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 (*sys.getfilesystemencoding()*).

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 (*sys.getfilesystemencoding()*). Returns 0 at EOF or a negative number upon failure.

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

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.

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.

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

struct _node* **PyParser_SimpleParseString** (const char *str, int start)

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

Deprecated since version 3.9, will be removed in version 3.10.

struct _node* **PyParser_SimpleParseStringFlags** (const char *str, int start, int flags)

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

Deprecated since version 3.9, will be removed in version 3.10.

struct _node* **PyParser_SimpleParseStringFlagsFilename** (const char *str, const char *filename, int start, int flags)

Parse Python source code from *str* using the start token *start* according to the *flags* argument. The result can be used to create a code object which can be evaluated efficiently. This is useful if a code fragment must be evaluated many times. *filename* is decoded from the filesystem encoding (`sys.getfilesystemencoding()`).

Deprecated since version 3.9, will be removed in version 3.10.

struct _node* **PyParser_SimpleParseFile** (FILE *fp, const char *filename, int start)

This is a simplified interface to `PyParser_SimpleParseFileFlags()` below, leaving *flags* set to 0.

Deprecated since version 3.9, will be removed in version 3.10.

struct _node* **PyParser_SimpleParseFileFlags** (FILE *fp, const char *filename, int start, int flags)

Similar to `PyParser_SimpleParseStringFlagsFilename()`, but the Python source code is read from *fp* instead of an in-memory string.

Deprecated since version 3.9, will be removed in version 3.10.

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

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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 (`sys.getfilesystemencoding()`). If *closeit* is true, the file is closed before *PyRun_FileExFlags()* returns.

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

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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).

3.4 sürümüyle geldi.

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

Return value: New reference. Like *Py_CompileStringObject()*, but *filename* is a byte string decoded from the filesystem encoding (`os.fsdecode()`).

3.2 sürümüyle geldi.

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

Return value: New reference. 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)

Return value: New reference. 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.

PyFrameObject

The C structure of the objects used to describe frame objects. The fields of this type are subject to change at any time.

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

Return value: New reference. Evaluate an execution frame. This is a simplified interface to *PyEval_EvalFrameEx()*, for backward compatibility.

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

Return value: New reference. 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.

3.4 sürümünde değişti: 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*.

3.8 sürümünde değişti: 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](#).

Reference Counting

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

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

Increment the reference count for object *o*. The object must not be NULL; if you aren't sure that it isn't NULL, use *Py_XINCREF* ().

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

Increment the reference count for object *o*. The object may be NULL, in which case the macro has no effect.

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

Decrement the reference count for object *o*. The object must not be NULL; if you aren't sure that it isn't NULL, use *Py_XDECREF* (). If the reference count reaches zero, the object's type's deallocation function (which must not be NULL) is invoked.

Uyarı: 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.

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

Decrement the reference count 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* (), and the same warning applies.

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

Decrement the reference count 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 decrementing its reference count.

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

The following functions are for runtime dynamic embedding of Python: *Py_IncRef* (*PyObject *o*), *Py_DecRef* (*PyObject *o*). They are simply exported function versions of *Py_XINCREF* () and *Py_XDECREF* (), respectively.

The following functions or macros are only for use within the interpreter core: `_Py_Dealloc()`, `_Py_ForgetReference()`, `_Py_NewReference()`, as well as the global variable `_Py_RefTotal`.

Exception Handling

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`).

When a function must fail because some function it called failed, it generally doesn't set the error indicator; the function it called already set it. It is responsible for either handling the error and clearing the exception or returning after cleaning up any resources it holds (such as object references or memory allocations); it should *not* continue normally if it is not prepared to handle the error. If returning due to an error, it is important to indicate to the caller that an error has been set. If the error is not handled or carefully propagated, additional calls into the Python/C API may not behave as intended and may fail in mysterious ways.

Not: 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**()

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

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

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 variables `sys.last_type`, `sys.last_value` and `sys.last_traceback` will be set to the type, value and traceback of the printed exception, respectively.

void **PyErr_Print** ()

Alias for `PyErr_PrintEx(1)`.

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

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.

An exception must be set when calling this function.

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)

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 increment its reference count. The second argument is an error message; it is decoded from 'utf-8'.

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

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, ...)

Return value: Always NULL. 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 args)

Return value: Always NULL. Same as `PyErr_Format()`, but taking a *va_list* argument rather than a variable number of arguments.

3.5 sürümüyle geldi.

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

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

int **PyErr_BadArgument** ()

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** ()

Return value: Always NULL. 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)

Return value: Always NULL. 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)

Return value: Always NULL. 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*)

Return value: Always NULL. Similar to `PyErr_SetFromErrnoWithFilenameObject()`, but takes a second filename object, for raising errors when a function that takes two filenames fails.

3.4 sürümüyle geldi.

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

Return value: Always NULL. Similar to `PyErr_SetFromErrnoWithFilenameObject()`, but the filename is given as a C string. *filename* is decoded from the filesystem encoding (`os.fsdecode()`).

***PyObject** PyErr_SetFromWindowsError** (int *ierr*)

Return value: Always NULL. This is a convenience function to raise `WindowsError`. 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 tuple object whose first item is the *ierr* value and whose second item is the corresponding error message (gotten from `FormatMessage()`), and then calls `PyErr_SetObject(PyExc_WindowsError, object)`. This function always returns NULL.

Availability: Windows.

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

Return value: Always NULL. Similar to `PyErr_SetFromWindowsError()`, with an additional parameter specifying the exception type to be raised.

Availability: Windows.

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

Return value: Always NULL. Similar to `PyErr_SetFromWindowsErrorWithFilenameObject()`, but the filename is given as a C string. *filename* is decoded from the filesystem encoding (`os.fsdecode()`).

Availability: Windows.

***PyObject** PyErr_SetExcFromWindowsErrorWithFilenameObject** (*PyObject* **type*, int *ierr*, *PyObject* **filenameObject*)

Return value: Always NULL. Similar to `PyErr_SetFromWindowsErrorWithFilenameObject()`, with an additional parameter specifying the exception type to be raised.

Availability: Windows.

***PyObject** PyErr_SetExcFromWindowsErrorWithFilenameObjects** (*PyObject* **type*, int *ierr*, *PyObject* **filenameObject*, *PyObject* **filenameObject2*)

Return value: Always NULL. Similar to `PyErr_SetExcFromWindowsErrorWithFilenameObject()`, but accepts a second filename object.

Availability: Windows.

3.4 sürümüyle geldi.

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

Return value: Always NULL. Similar to `PyErr_SetFromWindowsErrorWithFilename()`, with an additional parameter specifying the exception type to be raised.

Availability: Windows.

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

Return value: Always NULL. 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.

3.3 sürümüyle geldi.

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

Return value: Always *NULL*. Much like *PyErr_SetImportError()* but this function allows for specifying a subclass of *ImportError* to raise.

3.6 sürümüyle geldi.

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*.

3.4 sürümüyle geldi.

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

Like *PyErr_SyntaxLocationObject()*, but *filename* is a byte string decoded from the filesystem encoding (*os.fsdecode()*).

3.2 sürümüyle geldi.

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

Like *PyErr_SyntaxLocationEx()*, but the *col_offset* parameter is omitted.

void **PyErr_BadInternalCall** ()

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)

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.

3.4 sürümüyle geldi.

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

Similar to *PyErr_WarnExplicitObject()* except that *message* and *module* are UTF-8 encoded strings,

and *filename* is decoded from the filesystem encoding (`os.fsdecode()`).

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

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

3.2 sürümüyle geldi.

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

Function similar to `PyErr_WarnFormat()`, but *category* is `ResourceWarning` and it passes *source* to `warnings.WarningMessage()`.

3.6 sürümüyle geldi.

5.4 Querying the error indicator

*PyObject** **PyErr_Occurred** ()

Return value: Borrowed reference. 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.

Not: 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)

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)

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.

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

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.

Not: This function is normally only used by code that needs to catch exceptions or by code that needs to save and restore the error indicator temporarily, e.g.:

```
{
    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)

Set the error indicator from the three objects. If the error indicator is already set, it is cleared first. 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.)

Not: This function is normally only used by 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*)

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.

Not: 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);
}
```

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

Retrieve 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.

Not: 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.

3.3 sürümüyle geldi.

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

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. For general rules about the three arguments, see `PyErr_Restore()`.

Not: 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.

3.3 sürümüyle geldi.

5.5 Signal Handling

int PyErr_CheckSignals ()

This function interacts with Python's signal handling. 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. In all cases, the default effect for `SIGINT` is to raise the `KeyboardInterrupt` exception. If an exception is raised the error indicator is set and the function returns `-1`; otherwise the function returns `0`. The error indicator may or may not be cleared if it was previously set.

void PyErr_SetInterrupt ()

Simulate the effect of a `SIGINT` signal arriving. The next time `PyErr_CheckSignals ()` is called, the Python signal handler for `SIGINT` will be called.

If `SIGINT` isn't handled by Python (it was set to `signal.SIG_DFL` or `signal.SIG_IGN`), this function does nothing.

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.

3.5 sürümünde değişti: On Windows, the function now also supports socket handles.

5.6 Exception Classes

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

Return value: New reference. 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)

Return value: New reference. 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.

3.2 sürümüyle geldi.

5.7 Exception Objects

PyObject* PyException_GetTraceback (PyObject *ex)

Return value: New reference. Return the traceback associated with the exception as a new reference, as accessible from Python through `__traceback__`. If there is no traceback associated, this returns `NULL`.

int PyException_SetTraceback (PyObject *ex, PyObject *tb)

Set the traceback associated with the exception to `tb`. Use `Py_None` to clear it.

PyObject* PyException_GetContext (PyObject *ex)

Return value: New reference. 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 `__context__`. If there is no context associated, this returns `NULL`.

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

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*)

Return value: *New reference.* 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 `__cause__`.

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

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*.

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

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*)

Return value: *New reference.* Create a `UnicodeDecodeError` object with the attributes *encoding*, *object*, *length*, *start*, *end* and *reason*. *encoding* and *reason* are UTF-8 encoded strings.

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

Return value: *New reference.* Create a `UnicodeEncodeError` object with the attributes *encoding*, *object*, *length*, *start*, *end* and *reason*. *encoding* and *reason* are UTF-8 encoded strings.

3.3 sürümünden beri önerilmiyor: 3.11

`Py_UNICODE` is deprecated since Python 3.3. Please migrate to `PyObject_CallFunction(PyExc_UnicodeEncodeError, "sOnns", ...)`.

*PyObject** **PyUnicodeTranslateError_Create** (const *Py_UNICODE* **object*, *Py_ssize_t* *length*,
Py_ssize_t *start*, *Py_ssize_t* *end*, const char **reason*)

Return value: *New reference.* Create a `UnicodeTranslateError` object with the attributes *object*, *length*, *start*, *end* and *reason*. *reason* is a UTF-8 encoded string.

3.3 sürümünden beri önerilmiyor: 3.11

`Py_UNICODE` is deprecated since Python 3.3. Please migrate to `PyObject_CallFunction(PyExc_UnicodeTranslateError, "Onns", ...)`.

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

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

Return value: *New reference.* Return the *encoding* attribute of the given exception object.

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

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

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

Return value: *New reference.* 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*)

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*)

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*)

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*)

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*)

Return value: New reference. 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*)

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

5.9 Recursion Control

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*)

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()`. In 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.

3.9 sürümünde değişti: This function is now also available in the limited API.

void **Py_LeaveRecursiveCall** (void)

Ends a `Py_EnterRecursiveCall()`. Must be called once for each *successful* invocation of `Py_EnterRecursiveCall()`.

3.9 sürümünde değişti: 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*)

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*)

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

5.10 Standard Exceptions

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:

C Name	Python Name	Notes
<code>PyExc_BaseException</code>	<code>BaseException</code>	1
<code>PyExc_Exception</code>	<code>Exception</code>	1
<code>PyExc_ArithmeticError</code>	<code>ArithmeticError</code>	1
<code>PyExc_AssertionError</code>	<code>AssertionError</code>	
<code>PyExc_AttributeError</code>	<code>AttributeError</code>	
<code>PyExc_BlockingIOError</code>	<code>BlockingIOError</code>	
<code>PyExc_BrokenPipeError</code>	<code>BrokenPipeError</code>	
<code>PyExc_BufferError</code>	<code>BufferError</code>	
<code>PyExc_ChildProcessError</code>	<code>ChildProcessError</code>	
<code>PyExc_ConnectionAbortedError</code>	<code>ConnectionAbortedError</code>	
<code>PyExc_ConnectionError</code>	<code>ConnectionError</code>	
<code>PyExc_ConnectionRefusedError</code>	<code>ConnectionRefusedError</code>	
<code>PyExc_ConnectionResetError</code>	<code>ConnectionResetError</code>	
<code>PyExc_EOFError</code>	<code>EOFError</code>	
<code>PyExc_FileExistsError</code>	<code>FileExistsError</code>	
<code>PyExc_FileNotFoundError</code>	<code>FileNotFoundError</code>	
<code>PyExc_FloatingPointError</code>	<code>FloatingPointError</code>	
<code>PyExc_GeneratorExit</code>	<code>GeneratorExit</code>	
<code>PyExc_ImportError</code>	<code>ImportError</code>	
<code>PyExc_IndentationError</code>	<code>IndentationError</code>	
<code>PyExc_IndexError</code>	<code>IndexError</code>	
<code>PyExc_InterruptedError</code>	<code>InterruptedError</code>	
<code>PyExc_IsADirectoryError</code>	<code>IsADirectoryError</code>	
<code>PyExc_KeyError</code>	<code>KeyError</code>	
<code>PyExc_KeyboardInterrupt</code>	<code>KeyboardInterrupt</code>	
<code>PyExc_LookupError</code>	<code>LookupError</code>	1
<code>PyExc_MemoryError</code>	<code>MemoryError</code>	
<code>PyExc_ModuleNotFoundError</code>	<code>ModuleNotFoundError</code>	
<code>PyExc_NameError</code>	<code>NameError</code>	
<code>PyExc_NotADirectoryError</code>	<code>NotADirectoryError</code>	
<code>PyExc_NotImplementedError</code>	<code>NotImplementedError</code>	
<code>PyExc_OSError</code>	<code>OSError</code>	1
<code>PyExc_OverflowError</code>	<code>OverflowError</code>	
<code>PyExc_PermissionError</code>	<code>PermissionError</code>	
<code>PyExc_ProcessLookupError</code>	<code>ProcessLookupError</code>	
<code>PyExc_RecursionError</code>	<code>RecursionError</code>	
<code>PyExc_ReferenceError</code>	<code>ReferenceError</code>	
<code>PyExc_RuntimeError</code>	<code>RuntimeError</code>	
<code>PyExc_StopAsyncIteration</code>	<code>StopAsyncIteration</code>	
<code>PyExc_StopIteration</code>	<code>StopIteration</code>	
<code>PyExc_SyntaxError</code>	<code>SyntaxError</code>	

Devamı sonraki sayfada

Tablo 1 – önceki sayfadan devam

C Name	Python Name	Notes
PyExc_SystemError	SystemError	
PyExc_SystemExit	SystemExit	
PyExc_TabError	TabError	
PyExc_TimeoutError	TimeoutError	
PyExc_TypeError	TypeError	
PyExc_UnboundLocalError	UnboundLocalError	
PyExc_UnicodeDecodeError	UnicodeDecodeError	
PyExc_UnicodeEncodeError	UnicodeEncodeError	
PyExc_UnicodeError	UnicodeError	
PyExc_UnicodeTranslateError	UnicodeTranslateError	
PyExc_ValueError	ValueError	
PyExc_ZeroDivisionError	ZeroDivisionError	

3.3 sürümüyle geldi: 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](#).

3.5 sürümüyle geldi: PyExc_StopAsyncIteration and PyExc_RecursionError.

3.6 sürümüyle geldi: PyExc_ModuleNotFoundError.

These are compatibility aliases to PyExc_OSError:

C Name	Notes
PyExc_EnvironmentError	
PyExc_IOError	
PyExc_WindowsError	²

3.3 sürümünde değişti: These aliases used to be separate exception types.

Notes:

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:

C Name	Python Name	Notes
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	

¹ This is a base class for other standard exceptions.

² Only defined on Windows; protect code that uses this by testing that the preprocessor macro `MS_WINDOWS` is defined.

3.2 sürümüyle geldi: `PyExc_ResourceWarning`.

Notes:

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

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)

Return value: *New reference.* Return the file system representation for *path*. If the object is a `str` or `bytes` object, then its reference count is incremented. 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.

3.6 sürümüyle geldi.

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 global flag `Py_InteractiveFlag` is true, this function also returns true if the *filename* pointer is `NULL` or if the name is equal to one of the strings `'<stdin>'` or `'???'`.

void PyOS_BeforeFork ()

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.

Uyarı: The C `fork()` call should only be made from the “*main*” thread (of the “*main*” interpreter). The same is true for `PyOS_BeforeFork()`.

3.7 sürümüyle geldi.

void PyOS_AfterFork_Parent ()

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.

Uyarı: The C `fork()` call should only be made from the “*main*” thread (of the “*main*” interpreter). The same is true for `PyOS_AfterFork_Parent()`.

3.7 sürümüyle geldi.

void **PyOS_AfterFork_Child()**

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.

Uyarı: The C `fork()` call should only be made from the “*main*” thread (of the “*main*” interpreter). The same is true for `PyOS_AfterFork_Child()`.

3.7 sürümüyle geldi.

Ayrıca bkz.:

`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()**

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.

3.7 sürümünden beri önerilmiyor: This function is superseded by `PyOS_AfterFork_Child()`.

int **PyOS_CheckStack()**

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 Windows using the Microsoft Visual C++ compiler). `USE_STACKCHECK` will be defined automatically; you should never change the definition in your own code.

`PyOS_sighandler_t` **PyOS_getsig**(int *i*)

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` is a typedef alias for `void (*)(int)`.

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

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! `PyOS_sighandler_t` is a typedef alias for `void (*)(int)`.

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

Decode a byte string from the locale encoding with the surrogateescape error handler: undecodable bytes are decoded as characters in range U+DC80..U+DCFF. If a byte sequence can be decoded as a surrogate character, escape the bytes using the surrogateescape error handler instead of decoding them.

Encoding, highest priority to lowest priority:

- UTF-8 on macOS, Android, and VxWorks;
- UTF-8 on Windows if `Py_LegacyWindowsFSEncodingFlag` is zero;
- UTF-8 if the Python UTF-8 mode is enabled;
- ASCII if the `LC_CTYPE` locale is "C", `nl_langinfo(CODESET)` returns the ASCII encoding (or an alias), and `mbstowcs()` and `wcstombs()` functions uses the ISO-8859-1 encoding.
- the current locale encoding.

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.

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.

Ayrıca bkz.:

The `PyUnicode_DecodeFSDefaultAndSize()` and `PyUnicode_DecodeLocaleAndSize()` functions.

3.5 sürümüyle geldi.

3.7 sürümünde değişti: The function now uses the UTF-8 encoding in the UTF-8 mode.

3.8 sürümünde değişti: The function now uses the UTF-8 encoding on Windows if `Py_LegacyWindowsFSEncodingFlag` is zero;

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

Encode a wide character string to the locale encoding with the surrogateescape error handler: surrogate characters in the range U+DC80..U+DCFF are converted to bytes 0x80..0xFF.

Encoding, highest priority to lowest priority:

- UTF-8 on macOS, Android, and VxWorks;
- UTF-8 on Windows if `Py_LegacyWindowsFSEncodingFlag` is zero;
- UTF-8 if the Python UTF-8 mode is enabled;
- ASCII if the `LC_CTYPE` locale is "C", `nl_langinfo(CODESET)` returns the ASCII encoding (or an alias), and `mbstowcs()` and `wcstombs()` functions uses the ISO-8859-1 encoding.
- the current locale encoding.

The function uses the UTF-8 encoding in the Python UTF-8 mode.

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.

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

Ayrıca bkz.:

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

3.5 sürümüyle geldi.

3.7 sürümünde değişti: The function now uses the UTF-8 encoding in the UTF-8 mode.

3.8 sürümünde değişti: The function now uses the UTF-8 encoding on Windows if `Py_LegacyWindowsFSEncodingFlag` 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*)

Return value: Borrowed reference. 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*)

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** ()

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

void **PySys_AddWarnOption** (const wchar_t *s)

Append `s` to `sys.warnoptions`. This function must be called prior to `Py_Initialize()` in order to affect the warnings filter list.

void **PySys_AddWarnOptionUnicode** (*PyObject* *unicode)

Append `unicode` to `sys.warnoptions`.

Note: this function is not currently usable from outside the CPython implementation, as it must be called prior to the implicit import of `warnings` in `Py_Initialize()` to be effective, but can't be called until enough of the runtime has been initialized to permit the creation of Unicode objects.

void **PySys_SetPath** (const wchar_t *path)

Set `sys.path` to a list object of paths found in `path` which should be a list of paths separated with the platform's search path delimiter (`:` on Unix, `;` on Windows).

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

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, ...)

As `PySys_WriteStdout()`, but write to `sys.stderr` or `stderr` instead.

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

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

3.2 sürümüyle geldi.

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

As `PySys_FormatStdout()`, but write to `sys.stderr` or `stderr` instead.

3.2 sürümüyle geldi.

void **PySys_AddXOption** (const wchar_t *s)

Parse `s` as a set of `-X` options and add them to the current options mapping as returned by `PySys_GetXOptions()`. This function may be called prior to `Py_Initialize()`.

3.2 sürümüyle geldi.

PyObject ***PySys_GetXOptions** ()

Return value: Borrowed reference. Return the current dictionary of `-X` options, similarly to `sys._xoptions`. On error, `NULL` is returned and an exception is set.

3.2 sürümüyle geldi.

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

Raise an auditing event with any active hooks. Return zero for success and non-zero with an exception set on failure.

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 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.

3.8 sürümüyle geldi.

3.8.2 sürümünde değişti: Require `Py_ssize_t` for # format characters. Previously, an unavoidable deprecation warning was raised.

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 of type `int (*)(const char *event, PyObject *args, void *userData)`, where *args* is guaranteed to be a `PyTupleObject`. 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 a 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.

3.8 sürümüyle geldi.

6.3 Process Control

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

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.

3.9 sürümünde değişti: Log the function name automatically.

void **Py_Exit** (int *status*)

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.

3.6 sürümünde değişti: Errors from finalization no longer ignored.

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

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)

Return value: New reference. This is a simplified interface to `PyImport_ImportModuleEx()` below, leaving the `globals` and `locals` arguments set to `NULL` and `level` set to 0. When the `name` argument contains a dot (when it specifies a submodule of a package), the `fromlist` argument is set to the list `['*']` so that the return value is the named module rather than the top-level package containing it as would otherwise be the case. (Unfortunately, this has an additional side effect when `name` in fact specifies a subpackage instead of a submodule: the submodules specified in the package's `__all__` variable are loaded.) Return a new reference to the imported module, or `NULL` with an exception set on failure. A failing import of a module doesn't leave the module in `sys.modules`.

This function always uses absolute imports.

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

Return value: New reference. This function is a deprecated alias of `PyImport_ImportModule()`.

3.3 sürümünde değişti: 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.

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

Return value: New reference. 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)

Return value: New reference. 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.

3.3 sürümüne geldi.

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

Return value: New reference. Similar to `PyImport_ImportModuleLevelObject()`, but the name is a UTF-8 encoded string instead of a Unicode object.

3.3 sürümünde değişti: Negative values for `level` are no longer accepted.

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

Return value: New reference. 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)

Return value: New reference. 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_AddModuleObject** (PyObject *name)

Return value: Borrowed reference. Return the module object corresponding to a module name. The `name` ar-

gument 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 `NULL` with an exception set on failure.

Not: 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.

3.3 sürümüyle geldi.

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

Return value: Borrowed reference. Similar to `PyImport_AddModuleObject()`, but the name is a UTF-8 encoded string instead of a Unicode object.

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

Return value: New reference. 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()`.

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

Return value: New reference. 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)

Return value: New reference. 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.

3.3 sürümüyle geldi.

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

Return value: New reference. 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`.

3.2 sürümüyle geldi.

3.3 sürümünde değişti: Uses `imp.source_from_cache()` in calculating the source path if only the bytecode path is provided.

long **PyImport_GetMagicNumber** ()

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.

3.3 sürümünde değişti: Return value of `-1` upon failure.

`const char * PyImport_GetMagicTag ()`

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.

3.2 sürümüyle geldi.

*PyObject** `PyImport_GetModuleDict ()`

Return value: Borrowed reference. 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)`

Return value: New reference. 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.

3.7 sürümüyle geldi.

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

Return value: New reference. 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)`

Return value: New reference. 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.)

3.3 sürümüyle geldi.

3.4 sürümünde değişti: The `__file__` attribute is no longer set on the module.

`int PyImport_ImportFrozenModule (const char *name)`

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;
};
```

`const struct _frozen* PyImport_FrozenModules`

This pointer is initialized to point to an array of `struct _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))`

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. Each of these structures gives the name and initialization function for a module built into the interpreter. The name is an ASCII encoded string. Programs which embed Python may use an array of these structures in conjunction with

`PyImport_ExtendInittab()` to provide additional built-in modules. The structure is defined in `Include/import.h` as:

```
struct _inittab {
    const char *name;           /* ASCII encoded string */
    PyObject* (*initfunc) (void);
};
```

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*)

Return *value*: New reference. 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*)

Return *value*: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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 Parsing arguments and building values

These functions are useful when creating your own extensions functions and methods. Additional information and examples are available in 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 Parsing arguments

A format string consists of zero or more “format units.” A format unit describes one Python object; it is usually a single character or a parenthesized sequence of format units. With a few exceptions, a format unit that is not a parenthesized sequence normally corresponds to a single address argument to these functions. In the following description, the quoted form is the format unit; the entry in (round) parentheses is the Python object type that matches the format unit; and the entry in [square] brackets is the type of the C variable(s) whose address should be passed.

Strings and buffers

These formats allow accessing an object as a contiguous chunk of memory. You don't have to provide raw storage for the returned unicode or bytes area.

In general, when a format sets a pointer to a buffer, the buffer is managed by the corresponding Python object, and the buffer shares the lifetime of this object. You won't have to release any memory yourself. The only exceptions are *es*, *es#*, *et* and *et#*.

However, when a *Py_buffer* structure gets filled, the underlying buffer is locked 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).

Unless otherwise stated, buffers are not NUL-terminated.

Some formats require a read-only *bytes-like object*, and set a pointer instead of a buffer structure. They work by checking that the object's *PyBufferProcs.bf_releasebuffer* field is NULL, which disallows mutable objects such as *bytearray*.

Not: For all # variants of formats (*s#*, *y#*, etc.), the type of the length argument (int or *Py_ssize_t*) is controlled by defining the macro *PY_SSIZE_T_CLEAN* before including *Python.h*. If the macro was defined, length is a *Py_ssize_t* rather than an int. This behavior will change in a future Python version to only support *Py_ssize_t* and drop int support. It is best to always define *PY_SSIZE_T_CLEAN*.

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.

Not: 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*.

3.5 sürümünde değişti: 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 *, int or Py_ssize_t] Like `s*`, except that it doesn't accept mutable objects. 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 *, int or 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 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.

3.5 sürümünde değişti: 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 *, int or Py_ssize_t] This variant on `s#` doesn't accept Unicode objects, only bytes-like objects.

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) [const Py_UNICODE *] Convert a Python Unicode object to a C pointer to a NUL-terminated buffer of Unicode characters. You must pass the address of a `Py_UNICODE` pointer variable, which will be filled with the pointer to an existing Unicode buffer. Please note that the width of a `Py_UNICODE` character depends on compilation options (it is either 16 or 32 bits). The Python string must not contain embedded null code points; if it does, a `ValueError` exception is raised.

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

Deprecated since version 3.3, will be removed in version 3.12: Part of the old-style `Py_UNICODE` API; please migrate to using `PyUnicode_AsWideCharString()`.

u# (str) [const Py_UNICODE *, int or Py_ssize_t] This variant on `u` stores into two C variables, the first one a pointer to a Unicode data buffer, the second one its length. This variant allows null code points.

Deprecated since version 3.3, will be removed in version 3.12: Part of the old-style `Py_UNICODE` API; please migrate to using `PyUnicode_AsWideCharString()`.

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

Deprecated since version 3.3, will be removed in version 3.12: Part of the old-style `Py_UNICODE` API; please migrate to using `PyUnicode_AsWideCharString()`.

z# (str or None) [const Py_UNICODE *, int or Py_ssize_t] Like `u#`, but the Python object may also be `None`, in which case the `Py_UNICODE` pointer is set to `NULL`.

Deprecated since version 3.3, will be removed in version 3.12: Part of the old-style `Py_UNICODE` API; please migrate to using `PyUnicode_AsWideCharString()`.

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] This variant on `s` is used for encoding Unicode into a character buffer. It only works for encoded data without embedded NUL bytes.

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] Same as `es` except that byte string objects are passed through without recoding them. Instead, the implementation assumes that the byte string object uses the encoding passed in as parameter.

es# (str) [const char *encoding, char **buffer, int or Py_ssize_t *buffer_length] This variant on `s#` is used for encoding Unicode into a character buffer. Unlike the `es` format, this variant allows input data which contains NUL characters.

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.

There are two modes of operation:

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.

In both cases, `*buffer_length` is set to the length of the encoded data without the trailing NUL byte.

et# (str, bytes or bytearray) [const char *encoding, char **buffer, int or Py_ssize_t *buffer_length]
Same as `es#` except that byte string objects are passed through without recoding them. Instead, the implementation assumes that the byte string object uses the encoding passed in as parameter.

Numbers

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`.

3.3 sürümünde değişti: 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.

Other objects

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. The object's reference count is not increased. The pointer stored is not `NULL`.

O!(object) [PyObject*, 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.

3.1 sürümünde değişti: `Py_CLEANUP_SUPPORTED` was added.

p (bool) [int] Tests the value passed in for truth (a boolean predicate) 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.

3.3 sürümüyle geldi.

(items) (tuple) [matching-items] The object must be a Python sequence whose length is the number of format units in *items*. The C arguments must correspond to the individual format units in *items*. Format units for sequences may be nested.

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).

A few other characters have a meaning in a format string. These may not occur inside nested parentheses. They are:

| 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.

3.3 sürümüyle geldi.

: 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).

; The list of format units ends here; the string after the semicolon is used as the error message *instead* of the default error message. : and ; mutually exclude each other.

Note that any Python object references which are provided to the caller are *borrowed* references; do not decrement their reference count!

Additional arguments passed to these functions must be addresses of variables whose type is determined by the format string; these are used to store values from the input tuple. There are a few cases, as described in the list of format units above, where these parameters are used as input values; they should match what is specified for the corresponding format unit in that case.

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.

API Functions

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

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 args)

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 *keywords[], ...)

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. Empty names denote *positional-only parameters*. Returns true on success; on failure, it returns false and raises the appropriate exception.

3.6 sürümünde değişti: Added support for *positional-only parameters*.

int PyArg_VaParseTupleAndKeywords (*PyObject* *args, *PyObject* *kw, const char *format, char *keywords[], va_list vars)
 Identical to *PyArg_ParseTupleAndKeywords()*, except that it accepts a *va_list* rather than a variable number of arguments.

int PyArg_ValidateKeywordArguments (*PyObject* *)
 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.

3.2 sürümüyle geldi.

int PyArg_Parse (*PyObject* *args, const char *format, ...)
 Function used to deconstruct the argument lists of “old-style” functions — these are functions which use the METH_OLDARGS parameter parsing method, which has been removed in Python 3. This is not recommended for use in parameter parsing in new code, and most code in the standard interpreter has been modified to no longer use this for that purpose. It does remain a convenient way to decompose other tuples, however, and may continue to be used for that purpose.

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

    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)
```

6.6.2 Building values

***PyObject** Py_BuildValue** (const char *format, ...)

Return value: New reference. 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 *, int or 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 *, int or 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 *, int or 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 *, int or 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 *, int or 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 (except for its reference count, which 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 increment the reference count on the object. 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 args)

Return value: New reference. 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, ...)

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)

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` and `format != NULL`.

If the platform doesn't have `vsnprintf()` and the buffer size needed to avoid truncation exceeds *size* by more than 512 bytes, Python aborts with a `Py_FatalError()`.

The return value (*rv*) for these functions should be interpreted as follows:

- When $0 \leq rv < size$, the output conversion was successful and *rv* characters were written to *str* (excluding the trailing `'\0'` byte at `str[rv]`).
- When $rv \geq size$, 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.

double **PyOS_string_to_double** (const char *s, char **endptr, *PyObject* *overflow_exception)

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`.

3.1 sürümüyle geldi.

char* **PyOS_double_to_string** (double val, char format_code, int precision, int flags, int *ptype)

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_DTST_SIGN`, `Py_DTST_ADD_DOT_0`, or `Py_DTST_ALT`, or-ed together:

- `Py_DTST_SIGN` means to always precede the returned string with a sign character, even if *val* is non-negative.
- `Py_DTST_ADD_DOT_0` means to ensure that the returned string will not look like an integer.
- `Py_DTST_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()`.

3.1 sürümüyle geldi.

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 Reflection

*PyObject** **PyEval_GetBuiltins** (void)

Return value: Borrowed reference. 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)

Return value: Borrowed reference. Return a dictionary of the local variables in the current execution frame, or NULL if no frame is currently executing.

*PyObject** **PyEval_GetGlobals** (void)

Return value: Borrowed reference. Return a dictionary of the global variables in the current execution frame, or NULL if no frame is currently executing.

*PyFrameObject** **PyEval_GetFrame** (void)

Return value: Borrowed reference. Return the current thread state's frame, which is NULL if no frame is currently executing.

See also *PyThreadState_GetFrame()*.

*PyFrameObject** **PyFrame_GetBack** (*PyFrameObject* *frame)

Get the *frame* next outer frame.

Return a strong reference, or NULL if *frame* has no outer frame.

frame must not be NULL.

3.9 sürümüyle geldi.

*PyCodeObject** **PyFrame_GetCode** (*PyFrameObject* *frame)

Get the *frame* code.

Return a strong reference.

frame must not be NULL. The result (frame code) cannot be NULL.

3.9 sürümüyle geldi.

int **PyFrame_GetLineNumber** (*PyFrameObject* *frame)

Return the line number that *frame* is currently executing.

frame must not be NULL.

const char* **PyEval_GetFuncName** (*PyObject* *func)

Return the name of *func* if it is a function, class or instance object, else the name of *funcs* type.

const char* **PyEval_GetFuncDesc** (*PyObject* *func)

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*.

6.9 Codec registry and support functions

int **PyCodec_Register** (*PyObject* *search_function)

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_KnownEncoding** (const char *encoding)

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)

Return value: New reference. 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)

Return value: New reference. 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.

6.9.1 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)

Return value: New reference. Get an encoder function for the given *encoding*.

*PyObject** **PyCodec_Decoder** (const char *encoding)

Return value: New reference. Get a decoder function for the given *encoding*.

*PyObject** **PyCodec_IncrementalEncoder** (const char *encoding, const char *errors)

Return value: New reference. Get an `IncrementalEncoder` object for the given *encoding*.

*PyObject** **PyCodec_IncrementalDecoder** (const char *encoding, const char *errors)

Return value: New reference. Get an `IncrementalDecoder` object for the given *encoding*.

*PyObject** **PyCodec_StreamReader** (const char *encoding, *PyObject* *stream, const char *errors)

Return value: New reference. Get a `StreamReader` factory function for the given *encoding*.

*PyObject** **PyCodec_StreamWriter** (const char *encoding, *PyObject* *stream, const char *errors)

Return value: New reference. Get a `StreamWriter` factory function for the given *encoding*.

6.9.2 Registry API for Unicode encoding error handlers

int **PyCodec_RegisterError** (const char *name, *PyObject* *error)

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/unencodable 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)

Return value: New reference. 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)

Return value: Always NULL. Raise *exc* as an exception.

*PyObject** **PyCodec_IgnoreErrors** (*PyObject* *exc)

Return value: New reference. Ignore the unicode error, skipping the faulty input.

*PyObject** **PyCodec_ReplaceErrors** (*PyObject* *exc)

Return value: New reference. Replace the unicode encode error with ? or U+FFFD.

*PyObject** **PyCodec_XMLCharRefReplaceErrors** (*PyObject* *exc)

Return value: New reference. Replace the unicode encode error with XML character references.

*PyObject** **PyCodec_BackslashReplaceErrors** (*PyObject* *exc)

Return value: New reference. Replace the unicode encode error with backslash escapes (`\x`, `\u` and `\U`).

*PyObject** **PyCodec_NameReplaceErrors** (*PyObject* *exc)

Return value: New reference. Replace the unicode encode error with `\N{ . . . }` escapes.

3.5 sürümüyle geldi.

Abstract Objects Layer

The functions in this chapter interact with Python objects regardless of their type, or with wide classes of object types (e.g. all numerical types, or all sequence types). When used on object types for which they do not apply, they will raise a Python exception.

It is not possible to use these functions on objects that are not properly initialized, such as a list object that has been created by `PyList_New()`, but whose items have not been set to some non-NULL value yet.

7.1 Object Protocol

*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, increment the reference count of `NotImplemented` and return it).

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_HasAttr** (*PyObject* *o, *PyObject* *attr_name)

Returns `1` if *o* has the attribute *attr_name*, and `0` otherwise. This is equivalent to the Python expression `hasattr(o, attr_name)`. This function always succeeds.

Note that exceptions which occur while calling `__getattr__()` and `__getattribute__()` methods will get suppressed. To get error reporting use *PyObject_GetAttr()* instead.

int **PyObject_HasAttrString** (*PyObject* *o, const char *attr_name)

Returns `1` if *o* has the attribute *attr_name*, and `0` otherwise. This is equivalent to the Python expression `hasattr(o, attr_name)`. This function always succeeds.

Note that exceptions which occur while calling `__getattr__()` and `__getattribute__()` methods and creating a temporary string object will get suppressed. To get error reporting use *PyObject_GetAttrString()* instead.

*PyObject** **PyObject_GetAttr** (*PyObject* *o, *PyObject* *attr_name)

Return value: New reference. 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*.

*PyObject** **PyObject_GetAttrString** (*PyObject* *o, const char *attr_name)

Return value: New reference. 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*.

*PyObject** **PyObject_GenericGetAttr** (*PyObject* *o, *PyObject* *name)

Return value: New reference. 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)

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)

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, but this feature is deprecated in favour of using *PyObject_DelAttrString()*.

int **PyObject_GenericSetAttr** (*PyObject* *o, *PyObject* *name, *PyObject* *value)

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)

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)

Delete attribute named *attr_name*, for object *o*. Returns -1 on failure. This is the equivalent of the Python statement *del o.attr_name*.

*PyObject** **PyObject_GenericGetDict** (*PyObject* *o, void *context)

Return value: New reference. A generic implementation for the getter of a *__dict__* descriptor. It creates the dictionary if necessary.

3.3 sürümüyle geldi.

int **PyObject_GenericSetDict** (*PyObject* *o, *PyObject* *value, void *context)

A generic implementation for the setter of a *__dict__* descriptor. This implementation does not allow the dictionary to be deleted.

3.3 sürümüyle geldi.

*PyObject** **PyObject_RichCompare** (*PyObject* *o1, *PyObject* *o2, int opid)

Return value: New reference. 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)

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. Returns -1 on error, 0 if the result is false, 1 otherwise. This is the equivalent of the Python expression *o1 op o2*, where *op* is the operator corresponding to *opid*.

Not: If *o1* and *o2* are the same object, `PyObject_RichCompareBool()` will always return 1 for `Py_EQ` and 0 for `Py_NE`.

*PyObject** **PyObject_Repr** (*PyObject* **o*)

Return value: New reference. 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.

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

*PyObject** **PyObject_ASCII** (*PyObject* **o*)

Return value: New reference. 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*)

Return value: New reference. 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.

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

*PyObject** **PyObject_Bytes** (*PyObject* **o*)

Return value: New reference. 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*)

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*)

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*)

Compute and return the hash value of an object *o*. On failure, return -1. This is the equivalent of the Python expression `hash(o)`.

3.2 sürümünde değişti: 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*)

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*)

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*)

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*)

Return value: New reference. 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 increments the reference count of 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 the incremented reference count is needed.

int PyObject_TypeCheck (*PyObject* **o*, *PyTypeObject* **type*)

Return true if the object *o* is of type *type* or a subtype of *type*. Both parameters must be non-NULL.

Py_ssize_t PyObject_Size (*PyObject* **o*)

Py_ssize_t PyObject_Length (*PyObject* **o*)

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)`.

3.4 sürümüyle geldi.

PyObject* PyObject_GetItem (*PyObject* **o*, *PyObject* **key*)

Return value: New reference. 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*)

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*)

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*)

Return value: New reference. 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*)

Return value: New reference. 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.

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

3.9 sürümüyle geldi.

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:

Uyarı: A class supporting vectorcall **must** also implement *tp_call* with the same semantics.

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:

```
PyObject * (*vectorcallfunc) (PyObject *callable, PyObject *const *args, size_t nargsf, PyObject *kwnames)
```

- *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

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.

To call an object that implements vectorcall, use a *call API* function as with any other callable. `PyObject_Vectorcall()` will usually be most efficient.

Not: In CPython 3.8, the vectorcall API and related functions were available provisionally under names with a leading underscore: `_PyObject_Vectorcall`, `_Py_TPFLAGS_HAVE_VECTORCALL`, `_PyObject_VectorcallMethod`, `_PyVectorcall_Function`, `_PyObject_CallOneArg`, `_PyObject_CallMethodNoArgs`, `_PyObject_CallMethodOneArg`. Additionally, `PyObject_VectorcallDict` was available as `_PyObject_FastCallDict`. The old names are still defined as aliases of the new, non-underscored names.

Recursion Control

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*)

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.

This function is not part of the *limited API*.

3.8 sürümüyle geldi.

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`.

This function is not part of the *limited API*.

3.8 sürümüyle geldi.

*PyObject** **PyVectorcall_Call** (*PyObject *callable*, *PyObject *tuple*, *PyObject *dict*)

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*.

This function is not part of the *limited API*.

3.8 sürümüyle geldi.

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
<code>PyObject_Call()</code>	<code>PyObject *</code>	tuple	dict/NULL
<code>PyObject_CallNoArgs()</code>	<code>PyObject *</code>	—	—
<code>PyObject_CallOneArg()</code>	<code>PyObject *</code>	1 object	—
<code>PyObject_CallObject()</code>	<code>PyObject *</code>	tuple/NULL	—
<code>PyObject_CallFunction()</code>	<code>PyObject *</code>	format	—
<code>PyObject_CallMethod()</code>	<code>obj + char*</code>	format	—
<code>PyObject_CallFunctionObjArgs()</code>	<code>PyObject *</code>	variadic	—
<code>PyObject_CallMethodObjArgs()</code>	<code>obj + name</code>	variadic	—
<code>PyObject_CallMethodNoArgs()</code>	<code>obj + name</code>	—	—
<code>PyObject_CallMethodOneArg()</code>	<code>obj + name</code>	1 object	—
<code>PyObject_Vectorcall()</code>	<code>PyObject *</code>	vectorcall	vectorcall
<code>PyObject_VectorcallDict()</code>	<code>PyObject *</code>	vectorcall	dict/NULL
<code>PyObject_VectorcallMethod()</code>	<code>arg + name</code>	vectorcall	vectorcall

*PyObject** **PyObject_Call** (*PyObject **callable, *PyObject **args, *PyObject **kwargs)

Return value: New reference. 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)

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.

3.9 sürümüyle geldi.

*PyObject** **PyObject_CallOneArg** (*PyObject **callable, *PyObject **arg)

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.

This function is not part of the *limited API*.

3.9 sürümüyle geldi.

*PyObject** **PyObject_CallObject** (*PyObject **callable, *PyObject **args)

Return value: New reference. 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*, ...)

Return value: New reference. 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.

3.4 sürümünde değişti: The type of *format* was changed from `char *`.

PyObject* PyObject_CallMethod (*PyObject *obj*, const char **name*, const char **format*, ...)

Return value: New reference. 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.

3.4 sürümünde değişti: The types of *name* and *format* were changed from `char *`.

PyObject* PyObject_CallFunctionObjArgs (*PyObject *callable*, ...)

Return value: New reference. 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*, ...)

Return value: New reference. 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.

This function is not part of the *limited API*.

3.9 sürümüyle geldi.

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.

This function is not part of the *limited API*.

3.9 sürümüyle geldi.

PyObject* PyObject_Vectorcall (*PyObject *callable*, *PyObject *const *args*, size_t *nargsf*, *PyObject *kwnames*)

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.

This function is not part of the *limited API*.

3.9 sürümüyle geldi.

*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.

This function is not part of the *limited API*.

3.9 sürümüyle geldi.

*PyObject** **PyObject_VectorcallMethod** (*PyObject* *name, *PyObject* *const *args, size_t nargsf, *PyObject* *kwnames)

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.

This function is not part of the *limited API*.

3.9 sürümüyle geldi.

7.2.4 Call Support API

int **PyCallable_Check** (*PyObject* *o)

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)

Returns 1 if the object *o* provides numeric protocols, and false otherwise. This function always succeeds.

3.8 sürümünde değişti: Returns 1 if *o* is an index integer.

*PyObject** **PyNumber_Add** (*PyObject* *o1, *PyObject* *o2)

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. Returns the result of matrix multiplication on *o1* and *o2*, or *NULL* on failure. This is the equivalent of the Python expression `o1 @ o2`.

3.5 sürümüyle geldi.

*PyObject** **PyNumber_FloorDivide** (*PyObject* *o1, *PyObject* *o2)

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. Returns o on success, or NULL on failure. This is the equivalent of the Python expression `+o`.

*PyObject** **PyNumber_Absolute** (*PyObject* *o)

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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`.

3.5 sürümüyle geldi.

PyObject* PyNumber_InPlaceFloorDivide (PyObject *o1, PyObject *o2)

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. Returns the *o* converted to a Python int on success or NULL with a `TypeError` exception raised on failure.

PyObject* PyNumber_ToBase (PyObject *n, int base)

Return value: New reference. 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*)

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*)

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*)

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*)

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*)

Return value: New reference. 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*)

Return value: New reference. 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*)

Return value: New reference. 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*)

Return value: New reference. 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*)

Return value: New reference. 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*)

Return value: New reference. 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*)

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*)

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*)

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*)

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*)

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*)

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*)

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*)

Return value: New reference. 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*)

Return value: New reference. 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*)

Return value: New reference. 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*)

Return value: Borrowed reference. 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*)

Return value: New reference. 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*)

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*)

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)

Return value: New reference. Return element of *o* corresponding to the string *key* or NULL on failure. This is the equivalent of the Python expression `o[key]`. See also `PyObject_GetItem()`.

int **PyMapping_SetItemString** (*PyObject *o*, const char *key, *PyObject *v*)

Map the string *key* to the value *v* in object *o*. Returns `-1` on failure. This is the equivalent of the Python statement `o[key] = v`. See also `PyObject_SetItem()`. This function *does not* steal a reference to *v*.

int **PyMapping_DelItem** (*PyObject *o*, *PyObject *key*)

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]`. This is an alias of `PyObject_DelItem()`.

int **PyMapping_DelItemString** (*PyObject *o*, const char *key)

Remove the mapping for the string *key* from the object *o*. Return `-1` on failure. This is equivalent to the Python statement `del o[key]`.

int **PyMapping_HasKey** (*PyObject *o*, *PyObject *key*)

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.

Note that exceptions which occur while calling the `__getitem__()` method will get suppressed. To get error reporting use `PyObject_GetItem()` instead.

int **PyMapping_HasKeyString** (*PyObject *o*, const char *key)

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.

Note that exceptions which occur while calling the `__getitem__()` method and creating a temporary string object will get suppressed. To get error reporting use `PyMapping_GetItemString()` instead.

*PyObject** **PyMapping_Keys** (*PyObject *o*)

Return value: New reference. On success, return a list of the keys in object *o*. On failure, return NULL.

3.7 sürümünde değişti: Previously, the function returned a list or a tuple.

*PyObject** **PyMapping_Values** (*PyObject *o*)

Return value: New reference. On success, return a list of the values in object *o*. On failure, return NULL.

3.7 sürümünde değişti: Previously, the function returned a list or a tuple.

*PyObject** **PyMapping_Items** (*PyObject *o*)

Return value: New reference. 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.

3.7 sürümünde değişti: 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*)

Return true if the object *o* supports the iterator protocol. This function always succeeds.

*PyObject** **PyIter_Next** (*PyObject *o*)

Return value: New reference. Return the next value from the iteration *o*. The object must be an iterator (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);
}

Py_DECREF(iterator);

if (PyErr_Occurred()) {
    /* propagate error */
}
else {
    /* continue doing useful work */
}
```

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()`.

Py_buffer

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.

void *obj

A new reference to the exporting object. The reference is owned by the consumer and automatically 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`.

`const char *format`

A *NUL* 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 macro `PyBUF_MAX_NDIM` limits the maximum number of dimensions to 64. Exporters MUST respect this limit, consumers of multi-dimensional buffers SHOULD be able to handle up to `PyBUF_MAX_NDIM` dimensions.

`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 *n*th 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.

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.

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 can be |'d to any of the flags except *PyBUF_SIMPLE*. The latter already implies format B (unsigned bytes).

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	yes	yes	if needed
PyBUF_STRIDES	yes	yes	NULL
PyBUF_ND	yes	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	yes	yes	NULL	C
PyBUF_F_CONTIGUOUS	yes	yes	NULL	F
PyBUF_ANY_CONTIGUOUS	yes	yes	NULL	C or F
<i>PyBUF_ND</i>	yes	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	yes	yes	if needed	U	0	yes
PyBUF_FULL_RO	yes	yes	if needed	U	1 or 0	yes
PyBUF_RECORDS	yes	yes	NULL	U	0	yes
PyBUF_RECORDS_RO	yes	yes	NULL	U	1 or 0	yes
PyBUF_STRIDED	yes	yes	NULL	U	0	NULL
PyBUF_STRIDED_RO	yes	yes	NULL	U	1 or 0	NULL
PyBUF_CONTIG	yes	NULL	NULL	C	0	NULL
PyBUF_CONTIG_RO	yes	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 *itemsizes*, *ndim*, *shape* and *strides*.

If *ndim* == 0, the memory location pointed to by *buf* is interpreted as a scalar of size *itemsizes*. 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, itemsizes, 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
    """
    if offset % itemsizes:
        return False
    if offset < 0 or offset+itemsizes > memlen:
        return False
    if any(v % itemsizes 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+itemsizes <= 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];
```

(continues on next page)

(önceki sayfadan devam)

```

    if (suboffsets[i] >= 0 ) {
        pointer = *((char**)pointer) + suboffsets[i];
    }
}
return (void*)pointer;
}

```

7.7.4 Buffer-related functions

int **PyObject_CheckBuffer** (*PyObject* *obj)

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)

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 *PyExc_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)

Release the buffer *view* and decrement the reference count for *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)

Return the implied *itemsizes* from *format*. On error, raise an exception and return -1.

3.9 sürümüyle geldi.

int **PyBuffer_IsContiguous** (*Py_buffer* *view, char order)

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** (*Py_buffer* *view, *Py_ssize_t* *indices)

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** (*Py_buffer* *view, void *buf, *Py_ssize_t* len, char fort)

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, *Py_buffer* *src, *Py_ssize_t* len, char order)

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*.

void **PyBuffer_FillContiguousStrides** (int ndims, *Py_ssize_t* *shape, *Py_ssize_t* *strides, int item-size, char order)

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)

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 `PyExc_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.

7.8 Old Buffer Protocol

3.0 sürümünden beri önerilmiyor.

These functions were part of the “old buffer protocol” API in Python 2. In Python 3, this protocol doesn’t exist anymore but the functions are still exposed to ease porting 2.x code. They act as a compatibility wrapper around the *new buffer protocol*, but they don’t give you control over the lifetime of the resources acquired when a buffer is exported.

Therefore, it is recommended that you call `PyObject_GetBuffer()` (or the *y** or *w** *format codes* with the `PyArg_ParseTuple()` family of functions) to get a buffer view over an object, and `PyBuffer_Release()` when the buffer view can be released.

int **PyObject_AsCharBuffer** (*PyObject* *obj, const char **buffer, *Py_ssize_t* *buffer_len)

Returns a pointer to a read-only memory location usable as character-based input. The *obj* argument must support the single-segment character buffer interface. On success, returns 0, sets *buffer* to the memory location and *buffer_len* to the buffer length. Returns -1 and sets a `TypeError` on error.

int **PyObject_AsReadBuffer** (*PyObject* *obj, const void **buffer, *Py_ssize_t* *buffer_len)

Returns a pointer to a read-only memory location containing arbitrary data. The *obj* argument must support the single-segment readable buffer interface. On success, returns 0, sets *buffer* to the memory location and *buffer_len* to the buffer length. Returns -1 and sets a `TypeError` on error.

int **PyObject_CheckReadBuffer** (*PyObject* *o)

Returns 1 if *o* supports the single-segment readable buffer interface. Otherwise returns 0. This function always succeeds.

Note that this function tries to get and release a buffer, and exceptions which occur while calling corresponding functions will get suppressed. To get error reporting use `PyObject_GetBuffer()` instead.

int **PyObject_AsWriteBuffer** (*PyObject* *obj, void **buffer, *Py_ssize_t* *buffer_len)

Returns a pointer to a writable memory location. The *obj* argument must support the single-segment, character buffer interface. On success, returns 0, sets *buffer* to the memory location and *buffer_len* to the buffer length. Returns -1 and sets a `TypeError` on error.

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.

Uyarı: 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

PyTypeObject

The C structure of the objects used to describe built-in types.

PyTypeObject **PyType_Type**

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** ()

Clear the internal lookup cache. Return the current version tag.

unsigned long **PyType_GetFlags** (*PyTypeObject** *type*)

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.

3.2 sürümüyle geldi.

3.4 sürümünde değişti: The return type is now `unsigned long` rather than `long`.

void **PyType_Modified** (*PyTypeObject* *type)

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_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)

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)

Return value: New reference. 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* *kwargs)

Return value: New reference. 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)

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.

Not: 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.

void* **PyType_GetSlot** (*PyTypeObject* *type, int slot)

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.

An exception is raised if *type* is not a heap type.

3.4 sürümüyle geldi.

*PyObject** **PyType_GetModule** (*PyTypeObject* *type)

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.

3.9 sürümüyle geldi.

`void* PyType_GetModuleState (PyTypeObject *type)`

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.

3.9 sürümüyle geldi.

Creating Heap-Allocated Types

The following functions and structs are used to create *heap types*.

*PyObject** **PyType_FromModuleAndSpec** (*PyObject* *module, *PyType_Spec* *spec, *PyObject* *bases)

Return value: New reference. Creates and returns a heap type object from the *spec* (*Py_TPFLAGS_HEAPTYPE*).

If *bases* is a tuple, the created heap type contains all types contained in it as base types.

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.

3.9 sürümüyle geldi.

*PyObject** **PyType_FromSpecWithBases** (*PyType_Spec* *spec, *PyObject* *bases)

Return value: New reference. Equivalent to *PyType_FromModuleAndSpec(NULL, spec, bases)*.

3.3 sürümüyle geldi.

*PyObject** **PyType_FromSpec** (*PyType_Spec* *spec)

Return value: New reference. Equivalent to *PyType_FromSpecWithBases(spec, NULL)*.

PyType_Spec

Structure defining a type's behavior.

`const char* PyType_Spec.name`

Name of the type, used to set *PyTypeObject.tp_name*.

`int PyType_Spec.basicsize`

`int PyType_Spec.itemsize`

Size of the instance in bytes, used to set *PyTypeObject.tp_basicsize* and *PyTypeObject.tp_itemsize*.

`int PyType_Spec.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** **PyType_Spec.slots**

Array of *PyType_Slot* structures. Terminated by the special slot value `{0, NULL}`.

PyType_Slot

Structure defining optional functionality of a type, containing a slot ID and a value pointer.

`int PyType_Slot.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 fields cannot be set at all using *PyType_Spec* and *PyType_Slot*:

- *tp_dict*
- *tp_mro*
- *tp_cache*
- *tp_subclasses*
- *tp_weaklist*
- *tp_vectorcall*
- *tp_weaklistoffset* (see *PyMemberDef*)
- *tp_dictoffset* (see *PyMemberDef*)
- *tp_vectorcall_offset* (see *PyMemberDef*)

The following fields cannot be set using *PyType_Spec* and *PyType_Slot* under the limited API:

- *bf_getbuffer*
- *bf_releasebuffer*

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.

3.9 sürümünde değişti: Slots in *PyBufferProcs* may be set in the unlimited API.

void ***PyType_Slot.pfunc**

The desired value of the slot. In most cases, this is a pointer to a function.

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. It needs to be treated just like any other object with respect to reference counts.

Py_RETURN_NONE

Properly handle returning *Py_None* from within a C function (that is, increment the reference count of None and return it.)

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.

PyLongObject

This subtype of `PyObject` represents a Python integer object.

PyTypeObject PyLong_Type

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)

Return value: New reference. 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)

Return value: New reference. Return a new `PyLongObject` object from a C unsigned long, or NULL on failure.

PyObject* PyLong_FromSsize_t (Py_ssize_t v)

Return value: New reference. Return a new `PyLongObject` object from a C `Py_ssize_t`, or NULL on failure.

PyObject* PyLong_FromSize_t (size_t v)

Return value: New reference. Return a new `PyLongObject` object from a C `size_t`, or NULL on failure.

PyObject* PyLong_FromLongLong (long long v)

Return value: New reference. Return a new `PyLongObject` object from a C long long, or NULL on failure.

PyObject* PyLong_FromUnsignedLongLong (unsigned long long v)

Return value: New reference. Return a new `PyLongObject` object from a C unsigned long long, or NULL on failure.

PyObject* PyLong_FromDouble (double v)

Return value: New reference. 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)

Return value: New reference. Return a new `PyLongObject` based on the string value in `str`, which is interpreted according to the radix in `base`. If `pend` is non-NULL, `*pend` will point to the first character in `str` which follows the representation of the number. 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 spaces and single underscores after a base specifier and between digits are ignored. If there are no digits, `ValueError` will be raised.

PyObject* PyLong_FromUnicode (Py_UNICODE *u, Py_ssize_t length, int base)

Return value: New reference. Convert a sequence of Unicode digits to a Python integer value.

Deprecated since version 3.3, will be removed in version 3.10: Part of the old-style `Py_UNICODE` API; please migrate to using `PyLong_FromUnicodeObject()`.

*PyObject** **PyLong_FromUnicodeObject** (*PyObject* **u*, int *base*)

Return value: *New reference.* Convert a sequence of Unicode digits in the string *u* to a Python integer value.

3.3 sürümüyle geldi.

*PyObject** **PyLong_FromVoidPtr** (void **p*)

Return value: *New reference.* Create a Python integer from the pointer *p*. The pointer value can be retrieved from the resulting value using `PyLong_AsVoidPtr()`.

long **PyLong_AsLong** (*PyObject* **obj*)

Return a C long representation of *obj*. If *obj* is not an instance of `PyLongObject`, first call its `__index__()` or `__int__()` 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.

3.8 sürümünde değişti: Use `__index__()` if available.

3.8 sürümünden beri önerilmiyor: Using `__int__()` is deprecated.

long **PyLong_AsLongAndOverflow** (*PyObject* **obj*, int **overflow*)

Return a C long representation of *obj*. If *obj* is not an instance of `PyLongObject`, first call its `__index__()` or `__int__()` 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.

3.8 sürümünde değişti: Use `__index__()` if available.

3.8 sürümünden beri önerilmiyor: Using `__int__()` is deprecated.

long long **PyLong_AsLongLong** (*PyObject* **obj*)

Return a C long long representation of *obj*. If *obj* is not an instance of `PyLongObject`, first call its `__index__()` or `__int__()` 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.

3.8 sürümünde değişti: Use `__index__()` if available.

3.8 sürümünden beri önerilmiyor: Using `__int__()` is deprecated.

long long **PyLong_AsLongLongAndOverflow** (*PyObject* **obj*, int **overflow*)

Return a C long long representation of *obj*. If *obj* is not an instance of `PyLongObject`, first call its `__index__()` or `__int__()` 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.

3.2 sürümüyle geldi.

3.8 sürümünde değişti: Use `__index__()` if available.

3.8 sürümünden beri önerilmiyor: Using `__int__()` is deprecated.

Py_ssize_t **PyLong_AsSsize_t** (*PyObject* **pylong*)

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)

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)

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)

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.

3.1 sürümünde değişti: A negative *pylong* now raises *OverflowError*, not *TypeError*.

unsigned long **PyLong_AsUnsignedLongMask** (*PyObject* *obj)

Return a C unsigned long representation of *obj*. If *obj* is not an instance of *PyLongObject*, first call its *__index__()* or *__int__()* 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.

3.8 sürümünde değişti: Use *__index__()* if available.

3.8 sürümünden beri önerilmiyor: Using *__int__()* is deprecated.

unsigned long long **PyLong_AsUnsignedLongLongMask** (*PyObject* *obj)

Return a C unsigned long long representation of *obj*. If *obj* is not an instance of *PyLongObject*, first call its *__index__()* or *__int__()* 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.

3.8 sürümünde değişti: Use *__index__()* if available.

3.8 sürümünden beri önerilmiyor: Using *__int__()* is deprecated.

double **PyLong_AsDouble** (*PyObject* *pylong)

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)

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.

8.2.2 Boolean Objects

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.

int **PyBool_Check** (*PyObject* *o)

Return true if *o* is of type `PyBool_Type`. This function always succeeds.

*PyObject** **Py_False**

The Python `False` object. This object has no methods. It needs to be treated just like any other object with respect to reference counts.

*PyObject** **Py_True**

The Python `True` object. This object has no methods. It needs to be treated just like any other object with respect to reference counts.

Py_RETURN_FALSE

Return `Py_False` from a function, properly incrementing its reference count.

Py_RETURN_TRUE

Return `Py_True` from a function, properly incrementing its reference count.

*PyObject** **PyBool_FromLong** (long v)

Return value: New reference. Return a new reference to `Py_True` or `Py_False` depending on the truth value of *v*.

8.2.3 Floating Point Objects

PyFloatObject

This subtype of *PyObject* represents a Python floating point object.

PyTypeObject **PyFloat_Type**

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)

Return value: New reference. Create a *PyFloatObject* object based on the string value in *str*, or NULL on failure.

*PyObject** **PyFloat_FromDouble** (double v)

Return value: New reference. Create a *PyFloatObject* object from *v*, or NULL on failure.

double **PyFloat_AsDouble** (*PyObject* *pyfloat)

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.

3.8 sürümünde değişti: 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)

Return value: New reference. 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** ()

Return the maximum representable finite float *DBL_MAX* as C double.

double **PyFloat_GetMin** ()

Return the minimum normalized positive float *DBL_MIN* as C double.

8.2.4 Complex Number Objects

Python's complex number objects are implemented as two distinct types when viewed from the C API: one is the Python object exposed to Python programs, and the other is a C structure which represents the actual complex number value. The API provides functions for working with both.

Complex Numbers as C Structures

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.

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. It is defined as:

```
typedef struct {
    double real;
    double imag;
} Py_complex;
```

Py_complex **_Py_c_sum** (*Py_complex* left, *Py_complex* right)

Return the sum of two complex numbers, using the C *Py_complex* representation.

Py_complex **_Py_c_diff** (*Py_complex* left, *Py_complex* right)

Return the difference between two complex numbers, using the C *Py_complex* representation.

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)

Return the product of two complex numbers, using the C *Py_complex* representation.

Py_complex **_Py_c_quot** (*Py_complex* dividend, *Py_complex* divisor)

Return the quotient of two complex numbers, using the C *Py_complex* representation.

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

PyComplexObject

This subtype of *PyObject* represents a Python complex number object.

PyTypeObject **PyComplex_Type**

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)

Return value: New reference. Create a new Python complex number object from a C *Py_complex* value.

*PyObject** **PyComplex_FromDoubles** (double *real*, double *imag*)

Return value: New reference. Return a new *PyComplexObject* object from *real* and *imag*.

double **PyComplex_RealAsDouble** (*PyObject* *op)

Return the real part of *op* as a C double.

double **PyComplex_ImagAsDouble** (*PyObject* *op)

Return the imaginary part of *op* as a C double.

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 `-1.0` as a real value.

3.8 sürümünde değişti: 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.

PyBytesObject

This subtype of *PyObject* represents a Python bytes object.

PyTypeObject **PyBytes_Type**

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)

Return value: New reference. 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)

Return value: New reference. 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, ...)

Return value: New reference. 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
%%	<i>n/a</i>	The literal % character.
%c	int	A single byte, represented as a C int.
%d	int	Equivalent to <code>printf("%d").</code> ¹
%u	unsigned int	Equivalent to <code>printf("%u").</code> ¹
%ld	long	Equivalent to <code>printf("%ld").</code> ¹
%lu	unsigned long	Equivalent to <code>printf("%lu").</code> ¹
%zd	<i>Py_ssize_t</i>	Equivalent to <code>printf("%zd").</code> ¹
%zu	<i>size_t</i>	Equivalent to <code>printf("%zu").</code> ¹
%i	int	Equivalent to <code>printf("%i").</code> ¹
%x	int	Equivalent to <code>printf("%x").</code> ¹
%s	const char*	A null-terminated C character array.
%p	const void*	The hex representation of a C pointer. Mostly equivalent to <code>printf("%p")</code> except that it is guaranteed to start with the literal 0x regardless of what the platform's <code>printf</code> 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*)

Return value: New reference. Identical to *PyBytes_FromFormat()* except that it takes exactly two arguments.

*PyObject** **PyBytes_FromObject** (*PyObject* **o*)

Return value: New reference. Return the bytes representation of object *o* that implements the buffer protocol.

Py_ssize_t **PyBytes_Size** (*PyObject* **o*)

Return the length of the bytes in bytes object *o*.

Py_ssize_t **PyBytes_GET_SIZE** (*PyObject* **o*)

Macro form of *PyBytes_Size()* but without error checking.

char* **PyBytes_AsString** (*PyObject* **o*)

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*)

Macro form of *PyBytes_AsString()* but without error checking.

int **PyBytes_AsStringAndSize** (*PyObject* **obj*, char ***buffer*, *Py_ssize_t* **length*)

Return the null-terminated contents of the object *obj* through the output variables *buffer* and *length*.

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.

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*.

3.5 sürümünde değişti: Previously, *TypeError* was raised when embedded null bytes were encountered in the bytes object.

void **PyBytes_Concat** (*PyObject* ***bytes*, *PyObject* **newpart*)

Create a new bytes object in **bytes* containing the contents of *newpart* appended to *bytes*; the caller will own

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 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*)

Create a new bytes object in **bytes* containing the contents of *newpart* appended to *bytes*. This version decrements the reference count of *newpart*.

int **_PyBytes_Resize** (*PyObject **bytes*, *Py_ssize_t newsize*)

A way to resize a bytes object even though it is “immutable”. Only use this to build up a brand new bytes object; don’t use this if the bytes may already be known in other parts of the code. It is an error to call this function if the refcount on the input bytes object is not one. 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

PyByteArrayObject

This subtype of *PyObject* represents a Python bytearray object.

PyObject **PyByteArray_Type**

This instance of *PyObject* 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*)

Return value: New reference. Return a new bytearray object from any object, *o*, that implements the *buffer protocol*.

*PyObject** **PyByteArray_FromStringAndSize** (const char **string*, *Py_ssize_t len*)

Return value: New reference. Create a new bytearray object from *string* and its length, *len*. On failure, NULL is returned.

*PyObject** **PyByteArray_Concat** (*PyObject *a*, *PyObject *b*)

Return value: New reference. Concat bytearrays *a* and *b* and return a new bytearray with the result.

Py_ssize_t **PyByteArray_Size** (*PyObject *bytearray*)

Return the size of *bytearray* after checking for a NULL pointer.

char* **PyByteArray_AsString** (*PyObject *bytearray*)

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*)

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)`
Macro version of `PyByteArray_AsString()`.

`Py_ssize_t PyByteArray_GET_SIZE (PyObject *bytearray)`
Macro version of `PyByteArray_Size()`.

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).

`Py_UNICODE*` and UTF-8 representations are created on demand and cached in the Unicode object. The `Py_UNICODE*` representation is deprecated and inefficient.

Due to the transition between the old APIs and the new APIs, Unicode objects can internally be in two states depending on how they were created:

- “canonical” Unicode objects are all objects created by a non-deprecated Unicode API. They use the most efficient representation allowed by the implementation.
- “legacy” Unicode objects have been created through one of the deprecated APIs (typically `PyUnicode_FromUnicode()`) and only bear the `Py_UNICODE*` representation; you will have to call `PyUnicode_READY()` on them before calling any other API.

Not: The “legacy” Unicode object will be removed in Python 3.12 with deprecated APIs. All Unicode objects will be “canonical” since then. See **PEP 623** for more information.

Unicode Type

These are the basic Unicode object types used for the Unicode implementation in Python:

Py_UCS4

Py_UCS2

Py_UCS1

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`.

3.3 sürümüyle geldi.

Py_UNICODE

This is a typedef of `wchar_t`, which is a 16-bit type or 32-bit type depending on the platform.

3.3 sürümünde değişti: 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.

PyASCIIObject

PyCompactUnicodeObject

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.

3.3 sürümüyle geldi.

***PyObject* PyUnicode_Type**

This instance of *PyObject* represents the Python Unicode type. It is exposed to Python code as `str`.

The following APIs are really C macros and can be used to do fast checks and to access internal read-only data of Unicode objects:

int PyUnicode_Check (*PyObject* **o*)

Return true if the object *o* is a Unicode object or an instance of a Unicode subtype. This function always succeeds.

int PyUnicode_CheckExact (*PyObject* **o*)

Return true if the object *o* is a Unicode object, but not an instance of a subtype. This function always succeeds.

int PyUnicode_READY (*PyObject* **o*)

Ensure the string object *o* is in the “canonical” representation. This is required before using any of the access macros described below.

Returns 0 on success and -1 with an exception set on failure, which in particular happens if memory allocation fails.

3.3 sürümüyle geldi.

Deprecated since version 3.10, will be removed in version 3.12: This API will be removed with *PyUnicode_FromUnicode()*.

Py_ssize_t **PyUnicode_GET_LENGTH** (*PyObject* **o*)

Return the length of the Unicode string, in code points. *o* has to be a Unicode object in the “canonical” representation (not checked).

3.3 sürümüyle geldi.

*Py_UCS1** **PyUnicode_1BYTE_DATA** (*PyObject* **o*)

*Py_UCS2** **PyUnicode_2BYTE_DATA** (*PyObject* **o*)

*Py_UCS4** **PyUnicode_4BYTE_DATA** (*PyObject* **o*)

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 macro. Make sure *PyUnicode_READY()* has been called before accessing this.

3.3 sürümüyle geldi.

PyUnicode_WCHAR_KIND

PyUnicode_1BYTE_KIND

PyUnicode_2BYTE_KIND

PyUnicode_4BYTE_KIND

Return values of the *PyUnicode_KIND()* macro.

3.3 sürümüyle geldi.

Deprecated since version 3.10, will be removed in version 3.12: *PyUnicode_WCHAR_KIND* is deprecated.

unsigned int PyUnicode_KIND (*PyObject* **o*)

Return one of the *PyUnicode* kind constants (see above) that indicate how many bytes per character this Unicode object uses to store its data. *o* has to be a Unicode object in the “canonical” representation (not checked).

3.3 sürümüyle geldi.

void* **PyUnicode_DATA** (*PyObject* **o*)

Return a void pointer to the raw Unicode buffer. *o* has to be a Unicode object in the “canonical” representation (not checked).

3.3 sürümüyle geldi.

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 macro does not do any sanity checks and is intended for usage in loops. The caller should cache the *kind* value and *data* pointer

as obtained from other macro 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.

3.3 sürümüyle geldi.

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.

3.3 sürümüyle geldi.

Py_UCS4 **PyUnicode_READ_CHAR** (*PyObject* **o*, *Py_ssize_t* *index*)

Read a character from a Unicode object *o*, which must be in the “canonical” representation. This is less efficient than *PyUnicode_READ()* if you do multiple consecutive reads.

3.3 sürümüyle geldi.

PyUnicode_MAX_CHAR_VALUE (*o*)

Return the maximum code point that is suitable for creating another string based on *o*, which must be in the “canonical” representation. This is always an approximation but more efficient than iterating over the string.

3.3 sürümüyle geldi.

Py_ssize_t **PyUnicode_GET_SIZE** (*PyObject* **o*)

Return the size of the deprecated *Py_UNICODE* representation, in code units (this includes surrogate pairs as 2 units). *o* has to be a Unicode object (not checked).

Deprecated since version 3.3, will be removed in version 3.12: Part of the old-style Unicode API, please migrate to using *PyUnicode_GET_LENGTH()*.

Py_ssize_t **PyUnicode_GET_DATA_SIZE** (*PyObject* **o*)

Return the size of the deprecated *Py_UNICODE* representation in bytes. *o* has to be a Unicode object (not checked).

Deprecated since version 3.3, will be removed in version 3.12: Part of the old-style Unicode API, please migrate to using *PyUnicode_GET_LENGTH()*.

*Py_UNICODE** **PyUnicode_AS_UNICODE** (*PyObject* **o*)

const char* **PyUnicode_AS_DATA** (*PyObject* **o*)

Return a pointer to a *Py_UNICODE* representation of the object. The returned buffer is always terminated with an extra null code point. It may also contain embedded null code points, which would cause the string to be truncated when used in most C functions. The *AS_DATA* form casts the pointer to `const char *`. The *o* argument has to be a Unicode object (not checked).

3.3 sürümünde değişti: This macro is now inefficient – because in many cases the *Py_UNICODE* representation does not exist and needs to be created – and can fail (return `NULL` with an exception set). Try to port the code to use the new *PyUnicode_nBYTE_DATA()* macros or use *PyUnicode_WRITE()* or *PyUnicode_READ()*.

Deprecated since version 3.3, will be removed in version 3.12: Part of the old-style Unicode API, please migrate to using the *PyUnicode_nBYTE_DATA()* family of macros.

int **PyUnicode_IsIdentifier** (*PyObject* **o*)

Return 1 if the string is a valid identifier according to the language definition, section identifiers. Return 0 otherwise.

3.9 sürümünde değişti: 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.

3.3 sürümünden beri önerilmiyor: This function uses simple case mappings.

Py_UCS4 **Py_UNICODE_TOUPPER** (*Py_UCS4 ch*)

Return the character *ch* converted to upper case.

3.3 sürümünden beri önerilmiyor: This function uses simple case mappings.

Py_UCS4 **Py_UNICODE_TOTITLE** (*Py_UCS4 ch*)

Return the character *ch* converted to title case.

3.3 sürümünden beri önerilmiyor: This function uses simple case mappings.

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 macro 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 macro 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 macro does not raise exceptions.

These APIs can be used to work with surrogates:

Py_UNICODE_IS_SURROGATE (*ch*)

Check if *ch* is a surrogate ($0xD800 \leq ch \leq 0xDFFF$).

Py_UNICODE_IS_HIGH_SURROGATE (*ch*)

Check if *ch* is a high surrogate ($0xD800 \leq ch \leq 0xDBFF$).

Py_UNICODE_IS_LOW_SURROGATE (*ch*)

Check if *ch* is a low surrogate ($0xDC00 \leq ch \leq 0xDFFF$).

Py_UNICODE_JOIN_SURROGATES (*high*, *low*)

Join two surrogate characters and return a single *Py_UCS4* value. *high* and *low* are respectively the leading and trailing surrogates in a surrogate pair.

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*)

Return value: *New reference.* 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.

3.3 sürümüyle geldi.

*PyObject** **PyUnicode_FromKindAndData** (*int* *kind*, *const void* **buffer*, *Py_ssize_t* *size*)

Return value: *New reference.* 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*.

3.3 sürümüyle geldi.

*PyObject** **PyUnicode_FromStringAndSize** (*const char* **u*, *Py_ssize_t* *size*)

Return value: *New reference.* Create a Unicode object from the char buffer *u*. The bytes will be interpreted as being UTF-8 encoded. The buffer is copied into the new object. If the buffer is not *NULL*, the return value might be a shared object, i.e. modification of the data is not allowed.

If *u* is *NULL*, this function behaves like *PyUnicode_FromUnicode()* with the buffer set to *NULL*. This usage is deprecated in favor of *PyUnicode_New()*, and will be removed in Python 3.12.

*PyObject** **PyUnicode_FromString** (*const char* **u*)

Return value: *New reference.* Create a Unicode object from a UTF-8 encoded null-terminated char buffer *u*.

*PyObject** **PyUnicode_FromFormat** (*const char* **format*, ...)

Return value: *New reference.* 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. The following format characters are allowed:

Format Characters	Type	Comment
%%	<i>n/a</i>	The literal % character.
%c	int	A single character, represented as a C int.
%d	int	Equivalent to <code>printf("%d").</code> ¹
%u	unsigned int	Equivalent to <code>printf("%u").</code> ¹
%ld	long	Equivalent to <code>printf("%ld").</code> ¹
%li	long	Equivalent to <code>printf("%li").</code> ¹
%lu	unsigned long	Equivalent to <code>printf("%lu").</code> ¹
%lld	long long	Equivalent to <code>printf("%lld").</code> ¹
%lli	long long	Equivalent to <code>printf("%lli").</code> ¹
%llu	unsigned long long	Equivalent to <code>printf("%llu").</code> ¹
%zd	<code>Py_ssize_t</code>	Equivalent to <code>printf("%zd").</code> ¹
%zi	<code>Py_ssize_t</code>	Equivalent to <code>printf("%zi").</code> ¹
%zu	<code>size_t</code>	Equivalent to <code>printf("%zu").</code> ¹
%i	int	Equivalent to <code>printf("%i").</code> ¹
%x	int	Equivalent to <code>printf("%x").</code> ¹
%s	const char*	A null-terminated C character array.
%p	const void*	The hex representation of a C pointer. Mostly equivalent to <code>printf("%p")</code> except that it is guaranteed to start with the literal 0x regardless of what the platform's <code>printf</code> yields.
%A	PyObject*	The result of calling <code>ascii()</code> .
%U	PyObject*	A Unicode object.
%V	PyObject*, const char*	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 <code>PyObject_Str()</code> .
%R	PyObject*	The result of calling <code>PyObject_Repr()</code> .

An unrecognized format character causes all the rest of the format string to be copied as-is to the result string, and any extra arguments discarded.

Not: The width formatter unit is number of characters rather than bytes. The precision formatter unit is number of bytes 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).

3.2 sürümünde değişti: Support for "%lld" and "%llu" added.

3.3 sürümünde değişti: Support for "%li", "%lli" and "%zi" added.

3.4 sürümünde değişti: Support width and precision formatter for "%s", "%A", "%U", "%V", "%S", "%R" added.

PyObject* **PyUnicode_FromFormatV** (const char **format*, va_list *vargs*)

Return value: New reference. Identical to `PyUnicode_FromFormat()` except that it takes exactly two arguments.

PyObject* **PyUnicode_FromEncodedObject** (PyObject **obj*, const char **encoding*, const char **errors*)

Return value: New reference. 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.

¹ For integer specifiers (d, u, ld, li, lu, lld, lli, llu, zd, zi, zu, i, x): the 0-conversion flag has effect even when a precision is given.

Py_ssize_t **PyUnicode_GetLength** (*PyObject* *unicode)

Return the length of the Unicode object, in code points.

3.3 sürümüyle geldi.

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.

3.3 sürümüyle geldi.

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.

3.3 sürümüyle geldi.

int **PyUnicode_WriteChar** (*PyObject* *unicode, *Py_ssize_t* index, *Py_UCS4* character)

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 its reference count is one).

3.3 sürümüyle geldi.

Py_UCS4 **PyUnicode_ReadChar** (*PyObject* *unicode, *Py_ssize_t* index)

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 the macro version `PyUnicode_READ_CHAR()`.

3.3 sürümüyle geldi.

*PyObject** **PyUnicode_Substring** (*PyObject* *str, *Py_ssize_t* start, *Py_ssize_t* end)

Return value: New reference. Return a substring of `str`, from character index `start` (included) to character index `end` (excluded). Negative indices are not supported.

3.3 sürümüyle geldi.

*Py_UCS4** **PyUnicode_AsUCS4** (*PyObject* *u, *Py_UCS4* *buffer, *Py_ssize_t* buflen, int copy_null)

Copy the string `u` 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 `u`). `buffer` is returned on success.

3.3 sürümüyle geldi.

*Py_UCS4** **PyUnicode_AsUCS4Copy** (*PyObject* *u)

Copy the string `u` 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.

3.3 sürümüyle geldi.

Deprecated Py_UNICODE APIs

Deprecated since version 3.3, will be removed in version 3.12.

These API functions are deprecated with the implementation of [PEP 393](#). Extension modules can continue using them, as they will not be removed in Python 3.x, but need to be aware that their use can now cause performance and memory hits.

*PyObject** **PyUnicode_FromUnicode** (const *Py_UNICODE* **u*, *Py_ssize_t* *size*)

Return value: New reference. Create a Unicode object from the *Py_UNICODE* buffer *u* of the given size. *u* may be NULL which causes the contents to be undefined. It is the user's responsibility to fill in the needed data. The buffer is copied into the new object.

If the buffer is not NULL, the return value might be a shared object. Therefore, modification of the resulting Unicode object is only allowed when *u* is NULL.

If the buffer is NULL, *PyUnicode_READY()* must be called once the string content has been filled before using any of the access macros such as *PyUnicode_KIND()*.

Deprecated since version 3.3, will be removed in version 3.12: Part of the old-style Unicode API, please migrate to using *PyUnicode_FromKindAndData()*, *PyUnicode_FromWideChar()*, or *PyUnicode_New()*.

*Py_UNICODE** **PyUnicode_AsUnicode** (*PyObject* **unicode*)

Return a read-only pointer to the Unicode object's internal *Py_UNICODE* buffer, or NULL on error. This will create the *Py_UNICODE** representation of the object if it is not yet available. The buffer is always terminated with an extra null code point. Note that the resulting *Py_UNICODE* string may also contain embedded null code points, which would cause the string to be truncated when used in most C functions.

Deprecated since version 3.3, will be removed in version 3.12: Part of the old-style Unicode API, please migrate to using *PyUnicode_AsUCS4()*, *PyUnicode_AsWideChar()*, *PyUnicode_ReadChar()* or similar new APIs.

Deprecated since version 3.3, will be removed in version 3.10.

*PyObject** **PyUnicode_TransformDecimalToASCII** (*Py_UNICODE* **s*, *Py_ssize_t* *size*)

Return value: New reference. Create a Unicode object by replacing all decimal digits in *Py_UNICODE* buffer of the given *size* by ASCII digits 0–9 according to their decimal value. Return NULL if an exception occurs.

Deprecated since version 3.3, will be removed in version 3.11: Part of the old-style *Py_UNICODE* API; please migrate to using *Py_UNICODE_TODECIMAL()*.

*Py_UNICODE** **PyUnicode_AsUnicodeAndSize** (*PyObject* **unicode*, *Py_ssize_t* **size*)

Like *PyUnicode_AsUnicode()*, but also saves the *Py_UNICODE()* array length (excluding the extra null terminator) in *size*. Note that the resulting *Py_UNICODE** string may contain embedded null code points, which would cause the string to be truncated when used in most C functions.

3.3 sürümüyle geldi.

Deprecated since version 3.3, will be removed in version 3.12: Part of the old-style Unicode API, please migrate to using *PyUnicode_AsUCS4()*, *PyUnicode_AsWideChar()*, *PyUnicode_ReadChar()* or similar new APIs.

*Py_UNICODE** **PyUnicode_AsUnicodeCopy** (*PyObject* **unicode*)

Create a copy of a Unicode string ending with a null code point. Return NULL and raise a *MemoryError* exception on memory allocation failure, otherwise return a new allocated buffer (use *PyMem_Free()* to free the buffer). Note that the resulting *Py_UNICODE** string may contain embedded null code points, which would cause the string to be truncated when used in most C functions.

3.2 sürümüyle geldi.

Please migrate to using *PyUnicode_AsUCS4Copy()* or similar new APIs.

Py_ssize_t **PyUnicode_GetSize** (*PyObject* **unicode*)

Return the size of the deprecated *Py_UNICODE* representation, in code units (this includes surrogate pairs as 2 units).

Deprecated since version 3.3, will be removed in version 3.12: Part of the old-style Unicode API, please migrate to using `PyUnicode_GET_LENGTH()`.

*PyObject** **PyUnicode_FromObject** (*PyObject* *obj)

Return value: New reference. 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 the reference with incremented refcount.

Objects other than Unicode or its subtypes will cause a `TypeError`.

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* len, const char *errors)

Return value: New reference. 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 `Py_FileSystemDefaultEncoding` (the locale encoding read at Python startup).

This function ignores the Python UTF-8 mode.

Ayrıca bkz.:

The `Py_DecodeLocale()` function.

3.3 sürümüyle geldi.

3.7 sürümünde değişti: 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)

Return value: New reference. Similar to `PyUnicode_DecodeLocaleAndSize()`, but compute the string length using `strlen()`.

3.3 sürümüyle geldi.

*PyObject** **PyUnicode_EncodeLocale** (*PyObject* *unicode, const char *errors)

Return value: New reference. 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 `Py_FileSystemDefaultEncoding` (the locale encoding read at Python startup).

This function ignores the Python UTF-8 mode.

Ayrıca bkz.:

The `Py_EncodeLocale()` function.

3.3 sürümüyle geldi.

3.7 sürümünde değişti: 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

To encode and decode file names and other environment strings, `Py_FileSystemDefaultEncoding` should be used as the encoding, and `Py_FileSystemDefaultEncodeErrors` should be used as the 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)

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.

3.1 sürümüyle geldi.

3.6 sürümünde değişti: 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)

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.

3.2 sürümüyle geldi.

3.6 sürümünde değişti: Accepts a *path-like object*.

*PyObject** **PyUnicode_DecodeFSDefaultAndSize** (const char *s, *Py_ssize_t* size)

Return value: New reference. Decode a string using `Py_FileSystemDefaultEncoding` and the `Py_FileSystemDefaultEncodeErrors` error handler.

If `Py_FileSystemDefaultEncoding` is not set, fall back to the locale encoding.

`Py_FileSystemDefaultEncoding` is initialized at startup from the locale encoding and cannot be modified later. If you need to decode a string from the current locale encoding, use `PyUnicode_DecodeLocaleAndSize()`.

Ayrıca bkz.:

The `Py_DecodeLocale()` function.

3.6 sürümünde değişti: Use `Py_FileSystemDefaultEncodeErrors` error handler.

*PyObject** **PyUnicode_DecodeFSDefault** (const char *s)

Return value: New reference. Decode a null-terminated string using `Py_FileSystemDefaultEncoding` and the `Py_FileSystemDefaultEncodeErrors` error handler.

If `Py_FileSystemDefaultEncoding` is not set, fall back to the locale encoding.

Use `PyUnicode_DecodeFSDefaultAndSize()` if you know the string length.

3.6 sürümünde değişti: Use `Py_FileSystemDefaultEncodeErrors` error handler.

*PyObject** **PyUnicode_EncodeFSDefault** (*PyObject** unicode)

Return value: New reference. Encode a Unicode object to `Py_FileSystemDefaultEncoding` with the `Py_FileSystemDefaultEncodeErrors` error handler, and return bytes. Note that the resulting bytes object may contain null bytes.

If `Py_FileSystemDefaultEncoding` is not set, fall back to the locale encoding.

`Py_FileSystemDefaultEncoding` is initialized at startup from the locale encoding and cannot be modified later. If you need to encode a string to the current locale encoding, use `PyUnicode_EncodeLocale()`.

Ayrıca bkz.:

The `Py_EncodeLocale()` function.

3.2 sürümüyle geldi.

3.6 sürümünde değişti: Use `Py_FileSystemDefaultEncodeErrors` error handler.

wchar_t Support

`wchar_t` support for platforms which support it:

*PyObject** **PyUnicode_FromWideChar** (const `wchar_t` **w*, *Py_ssize_t* *size*)

Return value: New reference. Create a Unicode object from the `wchar_t` buffer *w* 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` **w*, *Py_ssize_t* *size*)

Copy the Unicode object contents into the `wchar_t` buffer *w*. 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. 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*)

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_Alloc()` (use `PyMem_Free()` to free it) on success. On error, returns `NULL` and **size* is undefined. Raises a `MemoryError` if memory allocation is failed.

3.2 sürümüyle geldi.

3.7 sürümünde değişti: 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 variable `Py_FileSystemDefaultEncoding` internally. This variable should be treated as read-only: on some systems, it will be a pointer to a static string, on others, it will change at run-time (such as when the application invokes `setlocale`).

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 *s, *Py_ssize_t* size, const char *encoding, const char *errors)
Return value: New reference. Create a Unicode object by decoding *size* bytes of the encoded string *s*. *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)
Return value: New reference. 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.

*PyObject** **PyUnicode_Encode** (const *Py_UNICODE* *s, *Py_ssize_t* size, const char *encoding, const char *errors)
Return value: New reference. Encode the *Py_UNICODE* buffer *s* of the given *size* and return a 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.

Deprecated since version 3.3, will be removed in version 3.11: Part of the old-style *Py_UNICODE* API; please migrate to using *PyUnicode_AsEncodedString()*.

UTF-8 Codecs

These are the UTF-8 codec APIs:

*PyObject** **PyUnicode_DecodeUTF8** (const char *s, *Py_ssize_t* size, const char *errors)
Return value: New reference. Create a Unicode object by decoding *size* bytes of the UTF-8 encoded string *s*. Return NULL if an exception was raised by the codec.

*PyObject** **PyUnicode_DecodeUTF8Stateful** (const char *s, *Py_ssize_t* size, const char *errors, *Py_ssize_t* *consumed)
Return value: New reference. 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)
Return value: New reference. 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.

const char* **PyUnicode_AsUTF8AndSize** (*PyObject* *unicode, *Py_ssize_t* *size)
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.

In the case of an error, NULL is returned with an exception set and no *size* is stored.

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.

3.3 sürümüyle geldi.

3.7 sürümünde değişti: The return type is now `const char *` rather of `char *`.

const char* **PyUnicode_AsUTF8** (*PyObject* *unicode)
As *PyUnicode_AsUTF8AndSize()*, but does not store the size.

3.3 sürümüyle geldi.

3.7 sürümünde değişti: The return type is now `const char *` rather of `char *`.

PyObject* PyUnicode_EncodeUTF8 (const *Py_UNICODE* *s, *Py_ssize_t* size, const char *errors)
Return value: New reference. Encode the *Py_UNICODE* buffer s of the given size using UTF-8 and return a Python bytes object. Return NULL if an exception was raised by the codec.

Deprecated since version 3.3, will be removed in version 3.11: Part of the old-style *Py_UNICODE* API; please migrate to using *PyUnicode_AsUTF8String()*, *PyUnicode_AsUTF8AndSize()* or *PyUnicode_AsEncodedString()*.

UTF-32 Codecs

These are the UTF-32 codec APIs:

PyObject* PyUnicode_DecompileUTF32 (const char *s, *Py_ssize_t* size, const char *errors, int *byteorder)
Return value: New reference. 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_DecompileUTF32Stateful (const char *s, *Py_ssize_t* size, const char *errors, int *byteorder, *Py_ssize_t* *consumed)
Return value: New reference. If *consumed* is NULL, behave like *PyUnicode_DecompileUTF32()*. If *consumed* is not NULL, *PyUnicode_DecompileUTF32Stateful()* 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)
Return value: New reference. 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.

PyObject* PyUnicode_EncodeUTF32 (const *Py_UNICODE* *s, *Py_ssize_t* size, const char *errors, int *byteorder*)
Return value: New reference. Return a Python bytes object holding the UTF-32 encoded value of the Unicode data in s. Output is written according to the following byte order:

```
byteorder == -1: little endian
byteorder == 0:  native byte order (writes a BOM mark)
byteorder == 1:  big endian
```

If *byteorder* is 0, the output string will always start with the Unicode BOM mark (U+FEFF). In the other two modes, no BOM mark is prepended.

If *Py_UNICODE_WIDE* is not defined, surrogate pairs will be output as a single code point.

Return NULL if an exception was raised by the codec.

Deprecated since version 3.3, will be removed in version 3.11: Part of the old-style *Py_UNICODE* API; please migrate to using *PyUnicode_AsUTF32String()* or *PyUnicode_AsEncodedString()*.

UTF-16 Codecs

These are the UTF-16 codec APIs:

*PyObject** **PyUnicode_DecodeUTF16** (const char *s, *Py_ssize_t* size, const char *errors, int *byteorder)
Return value: New reference. 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 \uffeff 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 *s, *Py_ssize_t* size, const char *errors, int *byteorder, *Py_ssize_t* *consumed)
Return value: New reference. 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)
Return value: New reference. 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.

*PyObject** **PyUnicode_EncodeUTF16** (const *Py_UNICODE* *s, *Py_ssize_t* size, const char *errors, int byteorder)
Return value: New reference. Return a Python bytes object holding the UTF-16 encoded value of the Unicode data in s. Output is written according to the following byte order:

```
byteorder == -1: little endian
byteorder == 0:  native byte order (writes a BOM mark)
byteorder == 1:  big endian
```

If byteorder is 0, the output string will always start with the Unicode BOM mark (U+FEFF). In the other two modes, no BOM mark is prepended.

If *Py_UNICODE_WIDE* is defined, a single *Py_UNICODE* value may get represented as a surrogate pair. If it is not defined, each *Py_UNICODE* value is interpreted as a UCS-2 character.

Return NULL if an exception was raised by the codec.

Deprecated since version 3.3, will be removed in version 3.11: Part of the old-style *Py_UNICODE* API; please migrate to using *PyUnicode_AsUTF16String()* or *PyUnicode_AsEncodedString()*.

UTF-7 Codecs

These are the UTF-7 codec APIs:

PyObject* **PyUnicode_DecodeUTF7** (const char *s, *Py_ssize_t* size, const char *errors)

Return value: New reference. Create a Unicode object by decoding *size* bytes of the UTF-7 encoded string *s*. Return NULL if an exception was raised by the codec.

PyObject* **PyUnicode_DecodeUTF7Stateful** (const char *s, *Py_ssize_t* size, const char *errors, *Py_ssize_t* *consumed)

Return value: New reference. 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*.

PyObject* **PyUnicode_EncodeUTF7** (const *Py_UNICODE* *s, *Py_ssize_t* size, int *base64SetO*, int *base64WhiteSpace*, const char *errors)

Return value: New reference. Encode the *Py_UNICODE* buffer of the given size using UTF-7 and return a Python bytes object. Return NULL if an exception was raised by the codec.

If *base64SetO* is nonzero, “Set O” (punctuation that has no otherwise special meaning) will be encoded in base-64. If *base64WhiteSpace* is nonzero, whitespace will be encoded in base-64. Both are set to zero for the Python “utf-7” codec.

Deprecated since version 3.3, will be removed in version 3.11: Part of the old-style *Py_UNICODE* API; please migrate to using `PyUnicode_AsEncodedString()`.

Unicode-Escape Codecs

These are the “Unicode Escape” codec APIs:

PyObject* **PyUnicode_DecodeUnicodeEscape** (const char *s, *Py_ssize_t* size, const char *errors)

Return value: New reference. Create a Unicode object by decoding *size* bytes of the Unicode-Escape encoded string *s*. Return NULL if an exception was raised by the codec.

PyObject* **PyUnicode_AsUnicodeEscapeString** (*PyObject* *unicode)

Return value: New reference. 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.

PyObject* **PyUnicode_EncodeUnicodeEscape** (const *Py_UNICODE* *s, *Py_ssize_t* size)

Return value: New reference. Encode the *Py_UNICODE* buffer of the given *size* using Unicode-Escape and return a bytes object. Return NULL if an exception was raised by the codec.

Deprecated since version 3.3, will be removed in version 3.11: Part of the old-style *Py_UNICODE* API; please migrate to using `PyUnicode_AsUnicodeEscapeString()`.

Raw-Unicode-Escape Codecs

These are the “Raw Unicode Escape” codec APIs:

PyObject* **PyUnicode_DecodeRawUnicodeEscape** (const char *s, *Py_ssize_t* size, const char *errors)

Return value: New reference. Create a Unicode object by decoding *size* bytes of the Raw-Unicode-Escape encoded string *s*. Return NULL if an exception was raised by the codec.

PyObject* **PyUnicode_AsRawUnicodeEscapeString** (*PyObject* *unicode)

Return value: New reference. 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.

PyObject* **PyUnicode_EncodeRawUnicodeEscape** (const *Py_UNICODE* *s, *Py_ssize_t* size)

Return value: New reference. Encode the *Py_UNICODE* buffer of the given *size* using Raw-Unicode-Escape and return a bytes object. Return NULL if an exception was raised by the codec.

Deprecated since version 3.3, will be removed in version 3.11: Part of the old-style `Py_UNICODE` API; please migrate to using `PyUnicode_AsRawUnicodeEscapeString()` or `PyUnicode_AsEncodedString()`.

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 *s, *Py_ssize_t* size, const char *errors)

Return value: New reference. Create a Unicode object by decoding *size* bytes of the Latin-1 encoded string *s*. Return NULL if an exception was raised by the codec.

*PyObject** **PyUnicode_AsLatin1String** (*PyObject* *unicode)

Return value: New reference. 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.

*PyObject** **PyUnicode_EncodeLatin1** (const *Py_UNICODE* *s, *Py_ssize_t* size, const char *errors)

Return value: New reference. Encode the `Py_UNICODE` buffer of the given *size* using Latin-1 and return a Python bytes object. Return NULL if an exception was raised by the codec.

Deprecated since version 3.3, will be removed in version 3.11: Part of the old-style `Py_UNICODE` API; please migrate to using `PyUnicode_AsLatin1String()` or `PyUnicode_AsEncodedString()`.

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 *s, *Py_ssize_t* size, const char *errors)

Return value: New reference. Create a Unicode object by decoding *size* bytes of the ASCII encoded string *s*. Return NULL if an exception was raised by the codec.

*PyObject** **PyUnicode_AsASCIIString** (*PyObject* *unicode)

Return value: New reference. 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.

*PyObject** **PyUnicode_EncodeASCII** (const *Py_UNICODE* *s, *Py_ssize_t* size, const char *errors)

Return value: New reference. Encode the `Py_UNICODE` buffer of the given *size* using ASCII and return a Python bytes object. Return NULL if an exception was raised by the codec.

Deprecated since version 3.3, will be removed in version 3.11: Part of the old-style `Py_UNICODE` API; please migrate to using `PyUnicode_AsASCIIString()` or `PyUnicode_AsEncodedString()`.

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 *data, *Py_ssize_t* size, *PyObject* *mapping, const char *errors)

Return value: New reference. Create a Unicode object by decoding *size* bytes of the encoded string *s* 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)

Return value: New reference. 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.

*PyObject** **PyUnicode_EncodeCharmap** (const *Py_UNICODE* *s, *Py_ssize_t* size, *PyObject* *mapping, const char *errors)

Return value: New reference. Encode the *Py_UNICODE* buffer of the given *size* using the given *mapping* object and return the result as a bytes object. Return NULL if an exception was raised by the codec.

Deprecated since version 3.3, will be removed in version 3.11: Part of the old-style *Py_UNICODE* API; please migrate to using *PyUnicode_AsCharmapString()* or *PyUnicode_AsEncodedString()*.

The following codec API is special in that maps Unicode to Unicode.

*PyObject** **PyUnicode_Translate** (*PyObject* *str, *PyObject* *table, const char *errors)

Return value: New reference. 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.

*PyObject** **PyUnicode_TranslateCharmap** (const *Py_UNICODE* *s, *Py_ssize_t* size, *PyObject* *mapping, const char *errors)

Return value: New reference. Translate a *Py_UNICODE* buffer of the given *size* by applying a character *mapping* table to it and return the resulting Unicode object. Return NULL when an exception was raised by the codec.

Deprecated since version 3.3, will be removed in version 3.11: Part of the old-style *Py_UNICODE* API; please migrate to using *PyUnicode_Translate()* or *generic codec based API*

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_DecompileMBCS** (const char *s, *Py_ssize_t* size, const char *errors)

Return value: New reference. Create a Unicode object by decoding *size* bytes of the MBCS encoded string *s*. Return NULL if an exception was raised by the codec.

*PyObject** **PyUnicode_DecompileMBCSStateful** (const char *s, *Py_ssize_t* size, const char *errors, *Py_ssize_t* *consumed)

Return value: New reference. If *consumed* is NULL, behave like *PyUnicode_DecompileMBCS()*. If *consumed* is not NULL, *PyUnicode_DecompileMBCSStateful()* 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)

Return value: New reference. 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)

Return value: New reference. 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.

3.3 sürümüyle geldi.

*PyObject** **PyUnicode_EncodeMBCS** (const *Py_UNICODE* *s, *Py_ssize_t* size, const char *errors)

Return value: New reference. Encode the *Py_UNICODE* buffer of the given *size* using MBCS and return a Python bytes object. Return NULL if an exception was raised by the codec.

Deprecated since version 3.3, will be removed in version 4.0: Part of the old-style *Py_UNICODE* API; please migrate to using *PyUnicode_AsMBCSString()*, *PyUnicode_EncodeCodePage()* or *PyUnicode_AsEncodedString()*.

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)

Return value: New reference. Concat two strings giving a new Unicode string.

*PyObject** **PyUnicode_Split** (*PyObject* *s, *PyObject* *sep, *Py_ssize_t* maxsplit)

Return value: New reference. 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* *s, int keepend)

Return value: New reference. Split a Unicode string at line breaks, returning a list of Unicode strings. CRLF is considered to be one line break. If *keepend* is 0, the line break characters are not included in the resulting strings.

*PyObject** **PyUnicode_Join** (*PyObject* *separator, *PyObject* *seq)

Return value: New reference. Join a sequence of strings using the given *separator* and return the resulting Unicode string.

Py_ssize_t **PyUnicode_Tailmatch** (*PyObject* *str, *PyObject* *substr, *Py_ssize_t* start, *Py_ssize_t* end, int direction)

Return 1 if *substr* matches *str*[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* *str, *PyObject* *substr, *Py_ssize_t* start, *Py_ssize_t* end, int direction)

Return the first position of *substr* in *str*[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* *str, *Py_UCS4* ch, *Py_ssize_t* start, *Py_ssize_t* end, int direction)

Return the first position of the character *ch* in *str*[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.

3.3 sürümüyle geldi.

3.7 sürümünde değişti: *start* and *end* are now adjusted to behave like *str*[start:end].

Py_ssize_t **PyUnicode_Count** (*PyObject* *str, *PyObject* *substr, *Py_ssize_t* start, *Py_ssize_t* end)

Return the number of non-overlapping occurrences of *substr* in *str*[start:end]. Return -1 if an error occurred.

*PyObject** **PyUnicode_Replace** (*PyObject* *str, *PyObject* *substr, *PyObject* *replstr, *Py_ssize_t* maxcount)

Return value: New reference. Replace at most *maxcount* occurrences of *substr* in *str* with *replstr* and return the

resulting Unicode object. *maxcount* == -1 means replace all occurrences.

int **PyUnicode_Compare** (*PyObject* *left, *PyObject* *right)

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_CompareWithASCIIString** (*PyObject* *uni, const char *string)

Compare a Unicode object, *uni*, 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)

Return value: New reference. 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)

Return value: New reference. Return a new string object from *format* and *args*; this is analogous to `format % args`.

int **PyUnicode_Contains** (*PyObject* *container, *PyObject* *element)

Check whether *element* is contained in *container* and return true or false accordingly.

element has to coerce to a one element Unicode string. -1 is returned if there was an error.

void **PyUnicode_InternInPlace** (*PyObject* **string)

Intern the argument **string* 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 **string*, it sets **string* to it (decrementing the reference count of the old string object and incrementing the reference count of the interned string object), otherwise it leaves **string* alone and interns it (incrementing its reference count). (Clarification: even though there is a lot of talk about reference counts, think of this function as reference-count-neutral; you own the object after the call if and only if you owned it before the call.)

*PyObject** **PyUnicode_InternFromString** (const char *v)

Return value: New reference. A combination of *PyUnicode_FromString()* and *PyUnicode_InternInPlace()*, returning either a new Unicode string object that has been interned, or a new (“owned”) reference to an earlier interned string object with the same value.

8.3.4 Tuple Objects

PyTupleObject

This subtype of *PyObject* represents a Python tuple object.

PyTypeObject PyTuple_Type

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)

Return value: New reference. Return a new tuple object of size *len*, or NULL on failure.

*PyObject** **PyTuple_Pack** (*Py_ssize_t* *n*, ...)

Return value: New reference. Return a new tuple object of size *n*, or NULL 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*)

Take a pointer to a tuple object, and return the size of that tuple.

Py_ssize_t **PyTuple_GET_SIZE** (*PyObject* **p*)

Return the size of the tuple *p*, which must be non-NULL and point to a tuple; no error checking is performed.

*PyObject** **PyTuple_GetItem** (*PyObject* **p*, *Py_ssize_t* *pos*)

Return value: Borrowed reference. Return the object at position *pos* in the tuple pointed to by *p*. If *pos* is out of bounds, return NULL and set an `IndexError` exception.

*PyObject** **PyTuple_GET_ITEM** (*PyObject* **p*, *Py_ssize_t* *pos*)

Return value: Borrowed reference. Like `PyTuple_GetItem()`, but does no checking of its arguments.

*PyObject** **PyTuple_GetSlice** (*PyObject* **p*, *Py_ssize_t* *low*, *Py_ssize_t* *high*)

Return value: New reference. Return the slice of the tuple pointed to by *p* between *low* and *high*, or NULL on failure. This is the equivalent of the Python expression `p[low:high]`. Indexing from the end of the list is not supported.

int **PyTuple_SetItem** (*PyObject* **p*, *Py_ssize_t* *pos*, *PyObject* **o*)

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.

Not: 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.

Not: This macro “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*)

Return value: New reference. Create a new struct sequence type from the data in *desc*, described below. Instances of the resulting type can be created with `PyStructSequence_New()`.

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*)

The same as `PyStructSequence_InitType`, but returns 0 on success and -1 on failure.

3.4 sürümüyle geldi.

PyStructSequence_Desc

Contains the meta information of a struct sequence type to create.

Field	C Type	Meaning
name	const char *	name of the struct sequence type
doc	const char *	pointer to docstring for the type or NULL to omit
fields	PyStructSequence_Field *	pointer to NULL-terminated array with field names of the new type
n_in_sequence	int	number of fields visible to the Python side (if used as tuple)

PyStructSequence_Field

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.

Field	C Type	Meaning
name	const char *	name for the field or NULL to end the list of named fields, set to <i>PyStructSequence_UnnamedField</i> to leave unnamed
doc	const char *	field docstring or NULL to omit

const char * const **PyStructSequence_UnnamedField**

Special value for a field name to leave it unnamed.

3.9 sürümünde değişti: The type was changed from char *.

*PyObject** **PyStructSequence_New** (*PyTypeObject* *type)

Return value: New reference. Creates an instance of *type*, which must have been created with *PyStructSequence_NewType()*.

*PyObject** **PyStructSequence_GetItem** (*PyObject* *p, *Py_ssize_t* pos)

Return value: Borrowed reference. Return the object at position *pos* in the struct sequence pointed to by *p*. No bounds checking is performed.

*PyObject** **PyStructSequence_GET_ITEM** (*PyObject* *p, *Py_ssize_t* pos)

Return value: Borrowed reference. Macro equivalent of *PyStructSequence_GetItem()*.

void **PyStructSequence_SetItem** (*PyObject* *p, *Py_ssize_t* pos, *PyObject* *o)

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.

Not: This function “steals” a reference to *o*.

void **PyStructSequence_SET_ITEM** (*PyObject* *p, *Py_ssize_t* *pos, *PyObject* *o)

Macro equivalent of *PyStructSequence_SetItem()*.

Not: This function “steals” a reference to *o*.

8.3.6 List Objects

PyListObject

This subtype of *PyObject* represents a Python list object.

PyTypeObject **PyList_Type**

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*)

Return value: New reference. Return a new list of length *len* on success, or NULL on failure.

Not: 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()*.

Py_ssize_t **PyList_Size** (*PyObject* **list*)

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*)

Macro form of *PyList_Size()* without error checking.

*PyObject** **PyList_GetItem** (*PyObject* **list*, *Py_ssize_t* *index*)

Return value: Borrowed reference. 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 $\geq \text{len}(\text{list})$), return NULL and set an `IndexError` exception.

*PyObject** **PyList_GET_ITEM** (*PyObject* **list*, *Py_ssize_t* *i*)

Return value: Borrowed reference. Macro form of *PyList_GetItem()* without error checking.

int PyList_SetItem (*PyObject* **list*, *Py_ssize_t* *index*, *PyObject* **item*)

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.

Not: 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.

Not: 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*)

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*)

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*)

Return value: New reference. 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)

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_Sort** (*PyObject* *list)

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)

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)

Return value: New reference. Return a new tuple object containing the contents of *list*; equivalent to `tuple(list)`.

8.4 Container Objects

8.4.1 Dictionary Objects

PyDictObject

This subtype of *PyObject* represents a Python dictionary object.

PyTypeObject **PyDict_Type**

This instance of *PyTypeObject* represents the Python dictionary type. This is the same object as `dict` in the Python layer.

int **PyDict_Check** (*PyObject* *p)

Return true if *p* is a dict object or an instance of a subtype of the dict type. This function always succeeds.

int **PyDict_CheckExact** (*PyObject* *p)

Return true if *p* is a dict object, but not an instance of a subtype of the dict type. This function always succeeds.

*PyObject** **PyDict_New** ()

Return value: New reference. Return a new empty dictionary, or `NULL` on failure.

*PyObject** **PyDictProxy_New** (*PyObject* *mapping)

Return value: New reference. 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)

Empty an existing dictionary of all key-value pairs.

int **PyDict_Contains** (*PyObject* *p, *PyObject* *key)

Determine if dictionary *p* contains *key*. If an item in *p* matches *key*, return 1, otherwise return 0. On error, return -1. This is equivalent to the Python expression `key in p`.

*PyObject** **PyDict_Copy** (*PyObject* *p)

Return value: New reference. Return a new dictionary that contains the same key-value pairs as *p*.

int **PyDict_SetItem** (*PyObject* *p, *PyObject* *key, *PyObject* *val)

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)

Insert *val* into the dictionary *p* using *key* as a key. *key* should be a `const char*`. The key object is created using `PyUnicode_FromString(key)`. Return 0 on success or -1 on failure. This function *does not* steal a reference to *val*.

int PyDict_DelItem (*PyObject* *p, *PyObject* *key)

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)

Remove the entry in dictionary *p* which has a key specified by the string *key*. If *key* is not in the dictionary, `KeyError` is raised. Return 0 on success or -1 on failure.

*PyObject** **PyDict_GetItem** (*PyObject* *p, *PyObject* *key)

Return value: Borrowed reference. Return the object from dictionary *p* which has a key *key*. Return NULL if the key *key* is not present, but *without* setting an exception.

Note that exceptions which occur while calling `__hash__()` and `__eq__()` methods will get suppressed. To get error reporting use `PyDict_GetItemWithError()` instead.

*PyObject** **PyDict_GetItemWithError** (*PyObject* *p, *PyObject* *key)

Return value: Borrowed reference. 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)

Return value: Borrowed reference. This is the same as `PyDict_GetItem()`, but *key* is specified as a const char*, rather than a *PyObject*.*.

Note that exceptions which occur while calling `__hash__()` and `__eq__()` methods and creating a temporary string object will get suppressed. To get error reporting use `PyDict_GetItemWithError()` instead.

*PyObject** **PyDict_SetDefault** (*PyObject* *p, *PyObject* *key, *PyObject* *defaultobj)

Return value: Borrowed reference. 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.

3.4 sürümüyle geldi.

*PyObject** **PyDict_Items** (*PyObject* *p)

Return value: New reference. Return a *PyListObject* containing all the items from the dictionary.

*PyObject** **PyDict_Keys** (*PyObject* *p)

Return value: New reference. Return a *PyListObject* containing all the keys from the dictionary.

*PyObject** **PyDict_Values** (*PyObject* *p)

Return value: New reference. Return a *PyListObject* containing all the values from the dictionary *p*.

Py_ssize_t **PyDict_Size** (*PyObject* *p)

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)

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.

For example:

```
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);
}
```

int **PyDict_Merge** (*PyObject* **a*, *PyObject* **b*, int *override*)

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*)

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*)

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
```

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()*).

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.

***PyObject* PySet_Type**

This is an instance of *PyObject* representing the Python set type.

***PyObject* PyFrozenSet_Type**

This is an instance of *PyObject* 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 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)**

Return value: New reference. 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)**

Return value: New reference. 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)**

Return the length of a set or frozenset object. Equivalent to `len(anyset)`. Raises a `PyExc_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)

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 `PyExc_SystemError` if *anyset* is not a set, frozenset, or an instance of a subtype.

int PySet_Add (*PyObject* *set, *PyObject* *key)

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)

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 `PyExc_SystemError` if *set* is not an instance of set or its subtype.

*PyObject** **PySet_Pop** (*PyObject *set*)

Return value: New reference. 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*)

Empty an existing set of all elements.

8.5 Function Objects

8.5.1 Function Objects

There are a few functions specific to Python functions.

PyFunctionObject

The C structure used for functions.

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*)

Return value: New reference. 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 function's name.

*PyObject** **PyFunction_NewWithQualName** (*PyObject *code*, *PyObject *globals*, *PyObject *qualname*)

Return value: New reference. 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 its `__name__` attribute.

3.3 sürümüyle geldi.

*PyObject** **PyFunction_GetCode** (*PyObject *op*)

Return value: Borrowed reference. Return the code object associated with the function object *op*.

*PyObject** **PyFunction_GetGlobals** (*PyObject *op*)

Return value: Borrowed reference. Return the globals dictionary associated with the function object *op*.

*PyObject** **PyFunction_GetModule** (*PyObject *op*)

Return value: Borrowed reference. Return the `__module__` attribute of the function object *op*. 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*)

Return value: Borrowed reference. 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.

*PyObject** **PyFunction_GetClosure** (*PyObject *op*)

Return value: Borrowed reference. 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*)

Return value: Borrowed reference. 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.

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*)

Return value: New reference. 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*)

Return value: Borrowed reference. Return the function object associated with the instance method *im*.

*PyObject ****PyInstanceMethod_GET_FUNCTION** (*PyObject *im*)

Return value: Borrowed reference. 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*)

Return value: New reference. 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*)

Return value: Borrowed reference. Return the function object associated with the method *meth*.

*PyObject ****PyMethod_GET_FUNCTION** (*PyObject *meth*)

Return value: Borrowed reference. Macro version of *PyMethod_Function()* which avoids error checking.

*PyObject ****PyMethod_Self** (*PyObject *meth*)

Return value: Borrowed reference. Return the instance associated with the method *meth*.

*PyObject** **PyMethod_GET_SELF** (*PyObject* *meth)

Return value: Borrowed reference. Macro version of *PyMethod_Self()* which avoids error checking.

8.5.4 Cell Objects

“Cell” objects are used to implement variables referenced by multiple scopes. For each such variable, a cell object is created to store the value; the local variables of each stack frame that references the value contains a reference to the cells from outer scopes which also use that variable. When the value is accessed, the value contained in the cell is used instead of the cell object itself. This de-referencing of the cell object requires support from the generated byte-code; these are not automatically de-referenced when accessed. Cell objects are not likely to be useful elsewhere.

PyCellObject

The C structure used for cell objects.

PyTypeObject **PyCell_Type**

The type object corresponding to cell objects.

int **PyCell_Check** (ob)

Return true if *ob* is a cell object; *ob* must not be NULL. This function always succeeds.

*PyObject** **PyCell_New** (*PyObject* *ob)

Return value: New reference. Create and return a new cell object containing the value *ob*. The parameter may be NULL.

*PyObject** **PyCell_Get** (*PyObject* *cell)

Return value: New reference. Return the contents of the cell *cell*.

*PyObject** **PyCell_GET** (*PyObject* *cell)

Return value: Borrowed reference. 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; if it is not a cell object, -1 will be returned. On success, 0 will be returned.

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.

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 type.

int **PyCode_Check** (*PyObject* *co)

Return true if *co* is a code object. This function always succeeds.

int **PyCode_GetNumFree** (*PyCodeObject* *co)

Return the number of free variables in *co*.

*PyCodeObject** **PyCode_New** (int argcount, int kwnonlyargcount, int nlocals, int stacksize, int flags, *PyObject* *code, *PyObject* *consts, *PyObject* *names, *PyObject* *varnames, *PyObject* *freevars, *PyObject* *cellvars, *PyObject* *filename, *PyObject* *name, int firstlineno, *PyObject* *lnotab)

Return value: New reference. Return a new code object. If you need a dummy code object to create a frame, use

`PyCode_NewEmpty()` instead. Calling `PyCode_New()` directly can bind you to a precise Python version since the definition of the bytecode changes often.

`PyObject* PyCode_NewWithPosOnlyArgs` (`int argcount`, `int posonlyargcount`, `int kwnonlyargcount`, `int nlocals`, `int stacksize`, `int flags`, `PyObject* code`, `PyObject* consts`, `PyObject* names`, `PyObject* varnames`, `PyObject* freevars`, `PyObject* cellvars`, `PyObject* filename`, `PyObject* name`, `int firstlineno`, `PyObject* notab`)

Return value: New reference. Similar to `PyCode_New()`, but with an extra “posonlyargcount” for positional-only arguments.

3.8 sürümüyle geldi.

`PyObject* PyCode_NewEmpty` (`const char* filename`, `const char* funcname`, `int firstlineno`)

Return value: New reference. Return a new empty code object with the specified filename, function name, and first line number. It is illegal to `exec()` or `eval()` the resulting code object.

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`)

Return value: New reference. 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.

Uyarı: 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).

3.2 sürümünde değişti: Ignore `name` attribute.

`int PyObject_AsFileDescriptor` (`PyObject* p`)

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`)

Return value: New reference. 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 `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.

This function is safe to call before `Py_Initialize()`.

Raises an auditing event `setopencodehook` with no arguments.

3.8 sürümüyle geldi.

int **PyFile_WriteObject** (*PyObject* *obj, *PyObject* *p, int flags)

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)

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**

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)

Return value: *New reference.* Return a new module object with the `__name__` attribute 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 providing a `__file__` attribute.

3.3 sürümüyle geldi.

3.4 sürümünde değişti: `__package__` and `__loader__` are set to None.

*PyObject** **PyModule_New** (const char *name)

Return value: *New reference.* Similar to `PyModule_NewObject()`, but the name is a UTF-8 encoded string instead of a Unicode object.

*PyObject** **PyModule_GetDict** (*PyObject* *module)

Return value: *Borrowed reference.* 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)

Return value: *New reference.* 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.

3.3 sürümüyle geldi.

const char* **PyModule_GetName** (*PyObject* *module)

Similar to `PyModule_GetNameObject()` but return the name encoded to 'utf-8'.

`void* PyModule_GetState (PyObject *module)`

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)`

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)`

Return value: *New reference.* 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 unicode string, raise `SystemError` and return NULL; otherwise return a reference to a Unicode object.

3.2 sürümüyle geldi.

`const char* PyModule_GetFilename (PyObject *module)`

Similar to `PyModule_GetFilenameObject ()` but return the filename encoded to 'utf-8'.

3.2 sürümünden beri önerilmiyor: `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.

`PyModuleDef`

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`.

3.5 sürümünde değişti: 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`.

3.9 sürümünde değişti: 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.

3.9 sürümünde değişti: 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`.

3.9 sürümünde değişti: 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)**

Return value: *New reference.* 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)**

Return value: *New reference.* 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.

Not: 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_AddObject()`.

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)

Return value: Borrowed reference. 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.

3.5 sürümüne geldi.

The `m_slots` member of the module definition must point to an array of `PyModuleDef_Slot` structures:

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.

3.5 sürümüne geldi.

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.

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)

Return value: New reference. Create a new module object, given the definition in *module* and the `ModuleSpec` *spec*. This behaves like `PyModule_FromDefAndSpec2()` with `module_api_version` set to `PYTHON_API_VERSION`.

3.5 sürümüyle geldi.

*PyObject** **PyModule_FromDefAndSpec2** (*PyModuleDef* *def, *PyObject* *spec, int module_api_version)

Return value: New reference. Create a new module object, given the definition in *module* 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.

Not: Most uses of this function should be using `PyModule_FromDefAndSpec()` instead; only use this if you are sure you need it.

3.5 sürümüyle geldi.

int **PyModule_ExecDef** (*PyObject* *module, *PyModuleDef* *def)

Process any execution slots (`Py_mod_exec`) given in *def*.

3.5 sürümüyle geldi.

int **PyModule_SetDocString** (*PyObject* *module, const char *docstring)

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`.

3.5 sürümüyle geldi.

int **PyModule_AddFunctions** (*PyObject* *module, *PyMethodDef* *functions)

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`.

3.5 sürümüyle geldi.

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_AddObject** (*PyObject* **module*, const char **name*, *PyObject* **value*)

Add an object to *module* as *name*. This is a convenience function which can be used from the module's initialization function. This steals a reference to *value* on success. Return -1 on error, 0 on success.

Not: Unlike other functions that steal references, `PyModule_AddObject()` only decrements the reference count of *value* **on success**.

This means that its return value must be checked, and calling code must `Py_DECREF()` *value* manually on error. Example usage:

```
Py_INCREF(spam);
if (PyModule_AddObject(module, "spam", spam) < 0) {
    Py_DECREF(module);
    Py_DECREF(spam);
    return NULL;
}
```

int **PyModule_AddIntConstant** (*PyObject* **module*, const char **name*, long *value*)

Add an integer constant to *module* as *name*. This convenience function can be used from the module's initialization function. Return -1 on error, 0 on success.

int **PyModule_AddStringConstant** (*PyObject* **module*, const char **name*, const char **value*)

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 on error, 0 on success.

int **PyModule_AddIntMacro** (*PyObject* **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 on error, 0 on success.

int **PyModule_AddStringMacro** (*PyObject* **module*, macro)

Add a string constant to *module*.

int **PyModule_AddType** (*PyObject* **module*, *PyTypeObject* **type*)

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 on error, 0 on success.

3.9 sürümüyle geldi.

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*)

Return value: Borrowed reference. 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*)

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 0 on success or -1 on failure.

3.3 sürümüyle geldi.

int **PyState_RemoveModule** (*PyModuleDef *def*)

Removes the module object created from *def* from the interpreter state. Return 0 on success or -1 on failure.

The caller must hold the GIL.

3.3 sürümüyle geldi.

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**

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** (*op*)

Return true if the type of *op* is `PySeqIter_Type`. This function always succeeds.

*PyObject** **PySeqIter_New** (*PyObject *seq*)

Return value: New reference. 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**

Type object for iterator objects returned by `PyCallIter_New()` and the two-argument form of the `iter()` built-in function.

int **PyCallIter_Check** (*op*)

Return true if the type of *op* is `PyCallIter_Type`. This function always succeeds.

*PyObject** **PyCallIter_New** (*PyObject *callable*, *PyObject *sentinel*)

Return value: New reference. 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**

The type object for the built-in descriptor types.

*PyObject** **PyDescr_NewGetSet** (*PyTypeObject *type*, struct *PyGetSetDef *getset*)

Return value: New reference.

*PyObject** **PyDescr_NewMember** (*PyTypeObject *type*, struct *PyMemberDef *meth*)

Return value: New reference.

*PyObject** **PyDescr_NewMethod** (*PyTypeObject *type*, struct *PyMethodDef *meth*)

Return value: New reference.

*PyObject** **PyDescr_NewWrapper** (*PyTypeObject* *type, struct wrapperbase *wrapper, void *wrapped)
Return value: New reference.

*PyObject** **PyDescr_NewClassMethod** (*PyTypeObject* *type, *PyMethodDef* *method)
Return value: New reference.

int **PyDescr_IsData** (*PyObject* *descr)
Return true if the descriptor objects *descr* describes a data attribute, or false if it describes a method. *descr* must be a descriptor object; there is no error checking.

*PyObject** **PyWrapper_New** (*PyObject* *, *PyObject* *)
Return value: New reference.

8.6.5 Slice Objects

PyTypeObject **PySlice_Type**
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)
Return value: New reference. 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 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)
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.

3.2 sürümünde değişti: 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)
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.

Returns 0 on success and -1 on error with exception set.

Not: 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);
```

3.2 sürümünde değişti: The parameter type for the *slice* parameter was `PySliceObject*` before.

3.6.1 sürümünde değişti: 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.

3.6.1 sürümünden beri önerilmiyor: 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)

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` on error, `0` on success.

3.6.1 sürümüyle geldi.

Py_ssize_t **PySlice_AdjustIndices** (*Py_ssize_t* length, *Py_ssize_t* *start, *Py_ssize_t* *stop, *Py_ssize_t* step)

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.

3.6.1 sürümüyle geldi.

8.6.6 Ellipsis Object

PyObject ***Py_Ellipsis**

The Python `Ellipsis` object. This object has no methods. It needs to be treated just like any other object with respect to reference counts. Like `Py_None` it is a singleton object.

8.6.7 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)

Return value: New reference. 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.

PyObject ***PyMemoryView_FromMemory** (char *mem, *Py_ssize_t* size, int flags)

Return value: New reference. Create a memoryview object using *mem* as the underlying buffer. *flags* can be one of `PyBUF_READ` or `PyBUF_WRITE`.

3.3 sürümüyle geldi.

PyObject ***PyMemoryView_FromBuffer** (*Py_buffer* *view)

Return value: New reference. 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)

Return value: New reference. Create a memoryview object to a *contiguous* chunk of memory (in either 'C' or 'Fortran 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.

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.

Py_buffer ***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.

8.6.8 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** (ob)

Return true if *ob* is either a reference or proxy object. This function always succeeds.

int **PyWeakref_CheckRef** (ob)

Return true if *ob* is a reference object. This function always succeeds.

int **PyWeakref_CheckProxy** (ob)

Return true if *ob* is a proxy object. This function always succeeds.

*PyObject** **PyWeakref_NewRef** (*PyObject* *ob, *PyObject* *callback)

Return value: New reference. 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-referencable object, or if *callback* is not callable, None, or NULL, this will return NULL and raise *TypeError*.

*PyObject** **PyWeakref_NewProxy** (*PyObject* *ob, *PyObject* *callback)

Return value: New reference. 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-referencable object, or if *callback* is not callable, None, or NULL, this will return NULL and raise *TypeError*.

*PyObject** **PyWeakref_GetObject** (*PyObject* *ref)

Return value: Borrowed reference. Return the referenced object from a weak reference, *ref*. If the referent is no longer live, returns *Py_None*.

Not: This function returns a **borrowed reference** to the referenced object. This means that you should always call *Py_INCREF()* on the object except if you know that it cannot be destroyed while you are still using it.

*PyObject** **PyWeakref_GET_OBJECT** (*PyObject* *ref)

Return value: Borrowed reference. Similar to *PyWeakref_GetObject()*, but implemented as a macro that does no error checking.

8.6.9 Capsules

Refer to using-capsules for more information on using these objects.

3.1 sürümüyle geldi.

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.

PyCapsule_Destructor

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*)

Return value: New reference. 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.attribute*. This will enable other modules to import the capsule using *PyCapsule_Import()*.

void* **PyCapsule_GetPointer** (*PyObject *capsule*, const char **name*)

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*)

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*)

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*)

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*)

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. If *no_block* is true, import the module without blocking (using *PyImport_ImportModuleNoBlock()*). If *no_block* is false, import the module conventionally (using *PyImport_ImportModule()*).

Return the capsule's internal *pointer* on success. On failure, set an exception and return NULL.

int **PyCapsule_IsValid** (*PyObject* *capsule, const char *name)

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)

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)

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)

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)

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.

8.6.10 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()*.

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)

Return value: New reference. 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)

Return value: New reference. 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

3.5 sürümüyle geldi.

Coroutine objects are what functions declared with an `async` keyword return.

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)

Return value: New reference. 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

Not: 3.7.1 sürümünde değişti: 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.

3.7 sürümüyle geldi.

This section details the public C API for the `contextvars` module.

PyContext

The C structure used to represent a `contextvars.Context` object.

PyContextVar

The C structure used to represent a `contextvars.ContextVar` object.

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)

Return value: New reference. Create a new empty context object. Returns NULL if an error has occurred.

PyObject ***PyContext_Copy** (*PyObject* *ctx)

Return value: New reference. Create a shallow copy of the passed *ctx* context object. Returns NULL if an error has occurred.

PyObject ***PyContext_CopyCurrent** (void)

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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.

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*.

3.7 sürümüyle geldi.

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)

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. Return a `datetime.datetime` object with the specified year, month, day, hour, minute, second, microsecond and fold.

3.6 sürümüyle geldi.

*PyObject** **PyTime_FromTime** (int hour, int minute, int second, int usecond)

Return value: New reference. 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)

Return value: New reference. Return a `datetime.time` object with the specified hour, minute, second, microsecond and fold.

3.6 sürümüyle geldi.

*PyObject** **PyDelta_FromDSU** (int *days*, int *seconds*, int *useconds*)

Return value: *New reference.* 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** (PyDateTime_DeltaType* *offset*)

Return value: *New reference.* Return a `datetime.timezone` object with an unnamed fixed offset represented by the *offset* argument.

3.7 sürümüyle geldi.

*PyObject** **PyTimeZone_FromOffsetAndName** (PyDateTime_DeltaType* *offset*, PyUnicode* *name*)

Return value: *New reference.* Return a `datetime.timezone` object with a fixed offset represented by the *offset* argument and with *tzname name*.

3.7 sürümüyle geldi.

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.

3.6 sürümüyle geldi.

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.

3.6 sürümüyle geldi.

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.

3.3 sürümüyle geldi.

int `PyDateTime_DELTA_GET_SECONDS` (`PyDateTime_Delta *o`)

Return the number of seconds, as an int from 0 through 86399.

3.3 sürümüyle geldi.

int `PyDateTime_DELTA_GET_MICROSECONDS` (`PyDateTime_Delta *o`)

Return the number of microseconds, as an int from 0 through 999999.

3.3 sürümüyle geldi.

Macros for the convenience of modules implementing the DB API:

***PyObject** `PyDateTime_FromTimestamp` (*PyObject* *args)**

Return value: New reference. Create and return a new `datetime.datetime` object given an argument tuple suitable for passing to `datetime.datetime.fromtimestamp()`.

***PyObject** `PyDate_FromTimestamp` (*PyObject* *args)**

Return value: New reference. 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. Only `GenericAlias` is exposed to C.

***PyObject** `Py_GenericAlias` (*PyObject* *origin, *PyObject* *args)**

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__", (PyCFunction)Py_GenericAlias, METH_O|METH_CLASS,
↪ "See PEP 585"}
    ...
}
```

Ayrıca bkz.:

The data model method `__class_getitem__()`.

3.9 sürümüyle geldi.

***PyTypeObject* `Py_GenericAliasType`**

The C type of the object returned by `Py_GenericAlias()`. Equivalent to `types.GenericAlias` in Python.

3.9 sürümüyle geldi.

Initialization, Finalization, and Threads

See also *Python Initialization Configuration*.

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:

- Configuration functions:

- `PyImport_AppendInittab()`
- `PyImport_ExtendInittab()`
- `PyInitFrozenExtensions()`
- `PyMem_SetAllocator()`
- `PyMem_SetupDebugHooks()`
- `PyObject_SetArenaAllocator()`
- `Py_SetPath()`
- `Py_SetProgramName()`
- `Py_SetPythonHome()`
- `Py_SetStandardStreamEncoding()`
- `PySys_AddWarnOption()`
- `PySys_AddXOption()`
- `PySys_ResetWarnOptions()`

- Informative functions:

- `Py_IsInitialized()`
- `PyMem_GetAllocator()`
- `PyObject_GetArenaAllocator()`

- `Py_GetBuildInfo()`
- `Py_GetCompiler()`
- `Py_GetCopyright()`
- `Py_GetPlatform()`
- `Py_GetVersion()`

- Utilities:

- `Py_DecodeLocale()`

- Memory allocators:

- `PyMem_RawMalloc()`
 - `PyMem_RawRealloc()`
 - `PyMem_RawCalloc()`
 - `PyMem_RawFree()`

Not: The following functions **should not be called** before `Py_Initialize()`: `Py_EncodeLocale()`, `Py_GetPath()`, `Py_GetPrefix()`, `Py_GetExecPrefix()`, `Py_GetProgramFullPath()`, `Py_GetPythonHome()`, `Py_GetProgramName()` and `PyEval_InitThreads()`.

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**

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.

int **Py_DebugFlag**

Turn on parser debugging output (for expert only, depending on compilation options).

Set by the `-d` option and the `PYTHONDEBUG` environment variable.

int **Py_DontWriteBytecodeFlag**

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.

int **Py_FrozenFlag**

Suppress error messages when calculating the module search path in `Py_GetPath()`.

Private flag used by `_freeze_importlib` and `frozenmain` programs.

int **Py_HashRandomizationFlag**

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.

int **Py_IgnoreEnvironmentFlag**

Ignore all `PYTHON*` environment variables, e.g. `PYTHONPATH` and `PYTHONHOME`, that might be set.

Set by the `-E` and `-I` options.

int `Py_InspectFlag`

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.

int `Py_InteractiveFlag`

Set by the `-i` option.

int `Py_IsolatedFlag`

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.

3.4 sürümüyle geldi.

int `Py_LegacyWindowsFSEncodingFlag`

If the flag is non-zero, use the `mbcs` encoding instead of the UTF-8 encoding for the filesystem encoding.

Set to 1 if the `PYTHONLEGACYWINDOWSFSENCODING` environment variable is set to a non-empty string.

See [PEP 529](#) for more details.

Availability: Windows.

int `Py_LegacyWindowsStdioFlag`

If the flag is non-zero, use `io.FileIO` instead of `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.

int `Py_NoSiteFlag`

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.

int `Py_NoUserSiteDirectory`

Don't add the user site-packages directory to `sys.path`.

Set by the `-s` and `-I` options, and the `PYTHONNOUSERSITE` environment variable.

int `Py_OptimizeFlag`

Set by the `-O` option and the `PYTHONOPTIMIZE` environment variable.

int `Py_QuietFlag`

Don't display the copyright and version messages even in interactive mode.

Set by the `-q` option.

3.2 sürümüyle geldi.

int `Py_UnbufferedStdioFlag`

Force the stdout and stderr streams to be unbuffered.

Set by the `-u` option and the `PYTHONUNBUFFERED` environment variable.

int `Py_VerboseFlag`

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.

9.3 Initializing and finalizing the interpreter

void **Py_Initialize**()

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 `PySys_SetArgvEx()` 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.

Not: 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*)

This function works like `Py_Initialize()` if *initsigs* is 1. If *initsigs* is 0, it skips initialization registration of signal handlers, which might be useful when Python is embedded.

int **Py_IsInitialized**()

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_FinalizeEx**()

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). 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.

3.6 sürümüyle geldi.

void **Py_Finalize**()

This is a backwards-compatible version of `Py_FinalizeEx()` that disregards the return value.

9.4 Process-wide parameters

int **Py_SetStandardStreamEncoding**(const char *encoding, const char *errors)

This function should be called before `Py_Initialize()`, if it is called at all. It specifies which encoding and error handling to use with standard IO, with the same meanings as in `str.encode()`.

It overrides `PYTHONIOENCODING` values, and allows embedding code to control IO encoding when the environment variable does not work.

encoding and/or *errors* may be `NULL` to use `PYTHONIOENCODING` and/or default values (depending on other settings).

Note that `sys.stderr` always uses the “backslashreplace” error handler, regardless of this (or any other) setting.

If `Py_FinalizeEx()` is called, this function will need to be called again in order to affect subsequent calls to `Py_Initialize()`.

Returns 0 if successful, a nonzero value on error (e.g. calling after the interpreter has already been initialized).

3.4 sürümüyle geldi.

void **Py_SetProgramName**(const wchar_t *name)

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_t*` string.

wchar_t* **Py_GetProgramName**()

Return the program name set with `Py_SetProgramName()`, or the default. The returned string points into static storage; the caller should not modify its value.

wchar_t* **Py_GetPrefix**()

Return the *prefix* for installed platform-independent files. This is derived through a number of complicated rules from the program name set with `Py_SetProgramName()` 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.

wchar_t* **Py_GetExecPrefix**()

Return the *exec-prefix* for installed platform-dependent files. This is derived through a number of complicated rules from the program name set with `Py_SetProgramName()` 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.

`wchar_t*` **Py_GetProgramFullPath()**

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 `Py_SetProgramName()` above). The returned string points into static storage; the caller should not modify its value. The value is available to Python code as `sys.executable`.

`wchar_t*` **Py_GetPath()**

Return the default module search path; this is computed from the program name (set by `Py_SetProgramName()` above) 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.

`void` **Py_SetPath** (const `wchar_t*`)

Set the default module search path. If this function is called before `Py_Initialize()`, then `Py_GetPath()` won't attempt to compute a default search path but uses the one provided instead. This is useful if Python is embedded by an application that has full knowledge of the location of all modules. The path components should be separated by the platform dependent delimiter character, which is `:` on Unix and macOS, `;` on Windows.

This also causes `sys.executable` to be set to the program full path (see `Py_GetProgramFullPath()`) and for `sys.prefix` and `sys.exec_prefix` to be empty. It is up to the caller to modify these if required after calling `Py_Initialize()`.

Use `Py_DecodeLocale()` to decode a bytes string to get a `wchar_t*` string.

The path argument is copied internally, so the caller may free it after the call completes.

3.8 sürümünde değişti: The program full path is now used for `sys.executable`, instead of the program name.

`const char*` **Py_GetVersion()**

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`.

`const char*` **Py_GetPlatform()**

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()**

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()**

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** ()

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*)

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_t*` string.

Not: 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");
```

3.1.3 sürümüyle geldi.

void **PySys_SetArgv** (int *argc*, wchar_t ***argv*)

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_t*` string.

3.4 sürümünde değişti: The *updatepath* value depends on `-I`.

void **Py_SetPythonHome** (const wchar_t **home*)

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_t*` string.

w_char* **Py_GetPythonHome** ()

Return the default “home”, that is, the value set by a previous call to `Py_SetPythonHome()`, or the value of the `PYTHONHOME` environment variable if it is set.

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.

Not: 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:

PyInterpreterState

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.

PyThreadState

This data structure represents the state of a single thread. The only public data member is `interp` (*PyInterpreterState **), which points to this thread's interpreter state.

void PyEval_InitThreads()

Deprecated function which does nothing.

In Python 3.6 and older, this function created the GIL if it didn't exist.

3.9 sürümünde değişti: The function now does nothing.

3.7 sürümünde değişti: This function is now called by *Py_Initialize()*, so you don't have to call it yourself anymore.

3.2 sürümünde değişti: This function cannot be called before *Py_Initialize()* anymore.

Deprecated since version 3.9, will be removed in version 3.11.

int PyEval_ThreadsInitialized()

Returns a non-zero value if *PyEval_InitThreads()* has been called. This function can be called without holding the GIL, and therefore can be used to avoid calls to the locking API when running single-threaded.

3.7 sürümünde değişti: The *GIL* is now initialized by *Py_Initialize()*.

Deprecated since version 3.9, will be removed in version 3.11.

*PyThreadState** **PyEval_SaveThread()**

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)

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.

Not: 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()**

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`).

*PyThreadState** **PyThreadState_Swap(PyThreadState *tstate)**

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()

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.

Not: 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)

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()

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.

3.4 sürümüyle geldi.

The following macros are normally used without a trailing semicolon; look for example usage in the Python source distribution.

Py_BEGIN_ALLOW_THREADS

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

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

This macro expands to `PyEval_RestoreThread(_save);`; it is equivalent to `Py_END_ALLOW_THREADS` without the closing brace.

Py_UNBLOCK_THREADS

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()`.

3.7 sürümünde değişti: `Py_Initialize()` now initializes the *GIL*.

*PyInterpreterState** **PyInterpreterState_New** ()

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)

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)

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)

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)

Reset all information in a thread state object. The global interpreter lock must be held.

3.9 sürümünde değişti: This function now calls the `PyThreadState.on_delete` callback. Previously, that happened in `PyThreadState_Delete()`.

void **PyThreadState_Delete** (*PyThreadState* *tstate)

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 need not be held. The thread state must have been reset with a previous call to `PyThreadState_Clear()`.

*PyFrameObject** **PyThreadState_GetFrame** (*PyThreadState* *tstate)

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.

3.9 sürümüyle geldi.

uint64_t **PyThreadState_GetID** (*PyThreadState* *tstate)

Get the unique thread state identifier of the Python thread state *tstate*.

tstate must not be NULL.

3.9 sürümüyle geldi.

*PyInterpreterState** **PyThreadState_GetInterpreter** (*PyThreadState* *tstate)

Get the interpreter of the Python thread state *tstate*.

tstate must not be NULL.

3.9 sürümüyle geldi.

*PyInterpreterState** **PyInterpreterState_Get** (void)

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.

3.9 sürümüyle geldi.

`int64_t PyInterpreterState_GetID (PyInterpreterState *interp)`

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.

3.7 sürümüyle geldi.

`PyObject* PyInterpreterState_GetDict (PyInterpreterState *interp)`

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.

3.8 sürümüyle geldi.

`PyObject* (*_PyFrameEvalFunction) (PyThreadState *tstate, PyFrameObject *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.

3.9 sürümünde değişti: The function now takes a `tstate` parameter.

`_PyFrameEvalFunction _PyInterpreterState_GetEvalFrameFunc (PyInterpreterState *interp)`

Get the frame evaluation function.

See the [PEP 523](#) “Adding a frame evaluation API to CPython”.

3.9 sürümüyle geldi.

`void _PyInterpreterState_SetEvalFrameFunc (PyInterpreterState *interp, _PyFrameEvalFunction eval_frame)`

Set the frame evaluation function.

See the [PEP 523](#) “Adding a frame evaluation API to CPython”.

3.9 sürümüyle geldi.

`PyObject* PyThreadState_GetDict ()`

Return value: Borrowed reference. 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)`

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.

3.7 sürümünde değişti: The type of the `id` parameter changed from `long` to `unsigned long`.

`void PyEval_AcquireThread (PyThreadState *tstate)`

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.

Not: 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.

3.8 sürümünde değişti: 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`)

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).

void **PyEval_AcquireLock** ()

Acquire the global interpreter lock. The lock must have been created earlier. If this thread already has the lock, a deadlock ensues.

3.2 sürümünden beri önerilmiyor: This function does not update the current thread state. Please use `PyEval_RestoreThread()` or `PyEval_AcquireThread()` instead.

Not: 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.

3.8 sürümünde değişti: 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.

void **PyEval_ReleaseLock** ()

Release the global interpreter lock. The lock must have been created earlier.

3.2 sürümünden beri önerilmiyor: This function does not update the current thread state. Please use `PyEval_SaveThread()` or `PyEval_ReleaseThread()` instead.

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:

`PyThreadState*` **Py_NewInterpreter** ()

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 return value points 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, `NULL` is returned; 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; however, unlike most other Python/C API functions, there needn't be a current thread state on entry.)

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 `inittestmodule` function is called again. As with multi-phase initialization, this means that only C-level static and global variables are shared between these modules.

void **Py_EndInterpreter** (*PyThreadState* *tstate)

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 must be held before calling this function and is still held when it returns.) `Py_FinalizeEx()` will destroy all sub-interpreters that haven't been explicitly destroyed at that point.

9.6.1 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*)

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.

Uyarı: 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*.

3.9 sürümünde değişti: 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.

3.1 sürümüyle geldi.

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.

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 <i>what</i>	Meaning of <i>arg</i>
<code>PyTrace_CALL</code>	Always <i>Py_None</i> .
<code>PyTrace_EXCEPTION</code>	Exception information as returned by <code>sys.exc_info()</code> .
<code>PyTrace_LINE</code>	Always <i>Py_None</i> .
<code>PyTrace_RETURN</code>	Value being returned to the caller, or <code>NULL</code> if caused by an exception.
<code>PyTrace_C_CALL</code>	Function object being called.
<code>PyTrace_C_EXCEPTION</code>	Function object being called.
<code>PyTrace_C_RETURN</code>	Function object being called.
<code>PyTrace_OPCODE</code>	Always <i>Py_None</i> .

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`.

The caller must hold the *GIL*.

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.

The caller must hold the *GIL*.

9.9 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.

9.10 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.

Not: 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.

9.10.1 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.

3.7 sürümüyle geldi.

Ayrıca bkz.:

“A New C-API for Thread-Local Storage in CPython” (PEP 539)

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** ()

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)

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`.

Not: 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)

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)

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)

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)

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)

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.

9.10.2 Thread Local Storage (TLS) API

3.7 sürümünden beri önerilmiyor: This API is superseded by *Thread Specific Storage (TSS) API*.

Not: 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** ()

void **PyThread_delete_key** (int key)

int **PyThread_set_key_value** (int key, void *value)

```
void* PyThread_get_key_value (int key)  
void PyThread_delete_key_value (int key)  
void PyThread_ReInitTLS ()
```

Python Initialization Configuration

3.8 sürümüyle geldi.

Structures:

- *PyConfig*
- *PyPreConfig*
- *PyStatus*
- *PyWideStringList*

Functions:

- *PyConfig_Clear()*
- *PyConfig_InitIsolatedConfig()*
- *PyConfig_InitPythonConfig()*
- *PyConfig_Read()*
- *PyConfig_SetArgv()*
- *PyConfig_SetBytesArgv()*
- *PyConfig_SetBytesString()*
- *PyConfig_SetString()*
- *PyConfig_SetWideStringList()*
- *PyPreConfig_InitIsolatedConfig()*
- *PyPreConfig_InitPythonConfig()*
- *PyStatus_Error()*
- *PyStatus_Exception()*
- *PyStatus_Exit()*
- *PyStatus_IsError()*
- *PyStatus_IsExit()*
- *PyStatus_NoMemory()*

- `PyStatus_Ok()`
- `PyWideStringList_Append()`
- `PyWideStringList_Insert()`
- `Py_ExitStatusException()`
- `Py_InitializeFromConfig()`
- `Py_PreInitialize()`
- `Py_PreInitializeFromArgs()`
- `Py_PreInitializeFromBytesArgs()`
- `Py_RunMain()`
- `Py_GetArgcArgv()`

The preconfiguration (`PyPreConfig` type) is stored in `_PyRuntime.preconfig` and the configuration (`PyConfig` type) is stored in `PyInterpreterState.config`.

See also *Initialization, Finalization, and Threads*.

Ayrıca bkz.:

PEP 587 “Python Initialization Configuration”.

10.1 PyWideStringList

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.

10.2 PyStatus

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.

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.

Not: 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)
{
```

(continues on next page)

(önceki sayfadan devam)

```
void *ptr;
PyStatus status = alloc(&ptr, 16);
if (PyStatus_Exception(status)) {
    Py_ExitStatusException(status);
}
PyMem_Free(ptr);
return 0;
}
```

10.3 PyPreConfig

PyPreConfig

Structure used to preinitialize Python:

- Set the Python memory allocator
- Configure the LC_CTYPE locale
- Set the UTF-8 mode

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 memory allocator:

- 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): force usage of `malloc()`
- 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_PYMALLOC and PYMEM_ALLOCATOR_PYMALLOC_DEBUG are not supported if Python is configured using `--without-pymalloc`

See *Memory Management*.

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` to 0.

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.

int **coerce_c_locale_warn**

If non-zero, emit a warning if the C locale is coerced.

int **dev_mode**

See *PyConfig.dev_mode*.

int isolated
See *PyConfig.isolated*.

int legacy_windows_fs_encoding (Windows only)
If non-zero, disable UTF-8 Mode, set the Python filesystem encoding to *mbcs*, set the filesystem error handler to *replace*.

Only available on Windows. `#ifdef MS_WINDOWS` macro can be used for Windows specific code.

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.

int use_environment
See *PyConfig.use_environment*.

int utf8_mode
If non-zero, enable the UTF-8 mode.

10.4 Preinitialization with PyPreConfig

Functions to preinitialize Python:

PyStatus **Py_PreInitialize** (const *PyPreConfig* **preconfig*)
Preinitialize Python from *preconfig* preconfiguration.

PyStatus **Py_PreInitializeFromBytesArgs** (const *PyPreConfig* **preconfig*, int *argc*, char * const **argv*)
Preinitialize Python from *preconfig* preconfiguration and command line arguments (bytes strings).

PyStatus **Py_PreInitializeFromArgs** (const *PyPreConfig* **preconfig*, int *argc*, wchar_t * const **argv*)
Preinitialize Python from *preconfig* preconfiguration and command line arguments (wide strings).

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 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 Python preinitialization, whereas calling directly *malloc()* and *free()* is always safe. *Py_DecodeLocale()* must not be called before the preinitialization.

Example using the preinitialization to enable the 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 will speak UTF-8 */
```

(continues on next page)

```
Py_Initialize();
/* ... use Python API here ... */
Py_Finalize();
```

10.5 PyConfig

PyConfig

Structure containing most parameters to configure Python.

Structure methods:

void **PyConfig_InitPythonConfig** (*PyConfig* **config*)

Initialize configuration with *Python Configuration*.

void **PyConfig_InitIsolatedConfig** (*PyConfig* **config*)

Initialize configuration with *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 from wide character strings.

Preinitialize Python if needed.

PyStatus **PyConfig_SetBytesArgv** (*PyConfig* **config*, int *argc*, char * const **argv*)

Set command line arguments: 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.

Preinitialize Python if needed.

void **PyConfig_Clear** (*PyConfig* **config*)

Release configuration memory.

Most `PyConfig` methods preinitialize Python if needed. In that case, the Python preinitialization configuration is based on the `PyConfig`. If configuration fields which are in common with `PyPreConfig` are tuned, they must be set before calling a `PyConfig` method:

- `dev_mode`
- `isolated`
- `parse_argv`
- `use_environment`

Moreover, if `PyConfig_SetArgv()` or `PyConfig_SetBytesArgv()` is used, this method must be called first, 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**

Command line arguments, `sys.argv`. See `parse_argv` to parse `argv` the same way the regular Python parses Python command line arguments. If `argv` is empty, an empty string is added to ensure that `sys.argv` always exists and is never empty.

wchar_t* **base_exec_prefix**
`sys.base_exec_prefix`.

wchar_t* **base_executable**
`sys._base_executable`: `__PYENV_LAUNCHER__` environment variable value, or copy of `PyConfig.executable`.

wchar_t* **base_prefix**
`sys.base_prefix`.

wchar_t* **platlibdir**
`sys.platlibdir`: platform library directory name, set at configure time by `--with-platlibdir`, overrideable by the `PYTHONPLATLIBDIR` environment variable.

3.9 sürümüyle geldi.

int **buffered_stdio**
 If equals to 0, enable unbuffered mode, making the stdout and stderr streams unbuffered.
 stdin is always opened in buffered mode.

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.

wchar_t* **check_hash_pycs_mode**
 Control the validation behavior of hash-based .pyc files (see **PEP 552**):
`--check-hash-based-pycs` command line option value.
 Valid values: always, never and default.
 The default value is: default.

int **configure_c_stdio**
 If non-zero, configure C standard streams (stdio, stdout, stdout). For example, set their mode to `O_BINARY` on Windows.

int **dev_mode**
 If non-zero, enable the Python Development Mode.

int **dump_refs**
 If non-zero, dump all objects which are still alive at exit.
`Py_TRACE_REFS` macro must be defined in build.

wchar_t* **exec_prefix**
`sys.exec_prefix`.

wchar_t* **executable**
`sys.executable`.

int **faulthandler**
 If non-zero, call `faulthandler.enable()` at startup.

wchar_t* **filesystem_encoding**

Filesystem encoding, `sys.getfilesystemencoding()`.

wchar_t* **filesystem_errors**

Filesystem encoding errors, `sys.getfilesystemencodeerrors()`.

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.

wchar_t* **home**

Python home directory.

Initialized from `PYTHONHOME` environment variable value by default.

int **import_time**

If non-zero, profile import time.

int **inspect**

Enter interactive mode after executing a script or a command.

int **install_signal_handlers**

Install signal handlers?

int **interactive**

Interactive mode.

int **isolated**

If greater than 0, enable isolated mode:

- `sys.path` contains neither the script's directory (computed from `argv[0]` or the current directory) nor the user's site-packages directory.
- Python REPL doesn't import `readline` nor enable default `readline` configuration on interactive prompts.
- Set `use_environment` and `user_site_directory` to 0.

int **legacy_windows_stdio**

If non-zero, use `io.FileIO` instead of `io.WindowsConsoleIO` for `sys.stdin`, `sys.stdout` and `sys.stderr`.

Only available on Windows. `#ifdef MS_WINDOWS` macro can be used for Windows specific code.

int **malloc_stats**

If non-zero, dump statistics on *Python pymalloc memory allocator* at exit.

The option is ignored if Python is built using `--without-pymalloc`.

wchar_t* **pythonpath_env**

Module search paths as a string separated by `DELIM(os.path.pathsep)`.

Initialized from `PYTHONPATH` environment variable value by default.

PyWideStringList **module_search_paths**

int **module_search_paths_set**

`sys.path`. If `module_search_paths_set` is equal to 0, the `module_search_paths` is overridden by the function calculating the *Path Configuration*.

int **optimization_level**

Compilation optimization level:

- 0: Peephole optimizer (and `__debug__` is set to `True`)
- 1: Remove assertions, set `__debug__` to `False`
- 2: Strip docstrings

int `parse_argv`

If non-zero, parse `argv` the same way the regular Python command line arguments, and strip Python arguments from `argv`: see Command Line Arguments.

int `parser_debug`

If non-zero, turn on parser debugging output (for expert only, depending on compilation options).

int `pathconfig_warnings`

If equal to 0, suppress warnings when calculating the *Path Configuration* (Unix only, Windows does not log any warning). Otherwise, warnings are written into `stderr`.

wchar_t* `prefix`

`sys.prefix`.

wchar_t* `program_name`

Program name. Used to initialize *executable*, and in early error messages.

wchar_t* `pycache_prefix`

`sys.pycache_prefix`: .pyc cache prefix.

If NULL, `sys.pycache_prefix` is set to None.

int `quiet`

Quiet mode. For example, don't display the copyright and version messages in interactive mode.

wchar_t* `run_command`

`python3 -c COMMAND` argument. Used by `Py_RunMain()`.

wchar_t* `run_filename`

`python3 FILENAME` argument. Used by `Py_RunMain()`.

wchar_t* `run_module`

`python3 -m MODULE` argument. Used by `Py_RunMain()`.

int `show_ref_count`

Show total reference count at exit?

Set to 1 by `-X showrefcount` command line option.

Need a debug build of Python (`Py_REF_DEBUG` macro must be defined).

int `site_import`

Import the `site` module at startup?

int `skip_source_first_line`

Skip the first line of the source?

wchar_t* `stdio_encoding`**wchar_t* `stdio_errors`**

Encoding and encoding errors of `sys.stdin`, `sys.stdout` and `sys.stderr`.

int `tracemalloc`

If non-zero, call `tracemalloc.start()` at startup.

int `use_environment`

If greater than 0, use environment variables.

int `user_site_directory`

If non-zero, add user site directory to `sys.path`.

int `verbose`

If non-zero, enable verbose mode.

***PyWideStringList* `warnoptions`**

`sys.warnoptions`: options of the warnings module to build warnings filters: lowest to highest priority.

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).

int **write_bytecode**

If non-zero, write `.pyc` files.

`sys.dont_write_bytecode` is initialized to the inverted value of `write_bytecode`.

PyWideStringList **xoptions**

`sys._xoptions`.

int **_use_peg_parser**

Enable PEG parser? Default: 1.

Set to 0 by `-X oldparser` and `PYTHONOLDPARSER`.

See also **PEP 617**.

Deprecated since version 3.9, will be removed in version 3.10.

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`: see [Command Line Arguments](#).

The `xoptions` options are parsed to set other options: see `-X` option.

3.9 sürümünde değişti: The `show_alloc_count` field has been removed.

10.6 Initialization with PyConfig

Function to initialize Python:

PyStatus **Py_InitializeFromConfig** (const *PyConfig* *config)

Initialize Python from *config* configuration.

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.

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 fail;
    }

    status = Py_InitializeFromConfig(&config);
    if (PyStatus_Exception(status)) {
        goto fail;
    }
    PyConfig_Clear(&config);
}
```

(continues on next page)

(önceki sayfadan devam)

```

    return;

fail:
    PyConfig_Clear(&config);
    Py_ExitStatusException(status);
}

```

More complete example modifying the default configuration, read the configuration, and then override some parameters:

```

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;
    }

    /* Append our custom search path to sys.path */
    status = PyWideStringList_Append(&config.module_search_paths,
                                     L"/path/to/more/modules");
    if (PyStatus_Exception(status)) {
        goto done;
    }

    /* 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;
}

```


10.7 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. Set the *Path Configuration* (“output fields”) to ignore these configuration files and avoid the function computing the default path configuration.

10.8 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 UTF-8 Mode (**PEP 540**) depending on the `LC_CTYPE` locale, `PYTHONUTF8` and `PYTHONCOERCECLOCALE` environment variables.

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);
    if (PyStatus_Exception(status)) {
        goto fail;
    }

    status = Py_InitializeFromConfig(&config);
    if (PyStatus_Exception(status)) {
        goto fail;
    }
    PyConfig_Clear(&config);

    return Py_RunMain();
fail:
    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);
}
```

10.9 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*)
 - `__PYENV_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, path configuration input fields are ignored as well.

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`
- `python._pth` (Windows only)
- `pybuilddir.txt` (Unix only)

The `__PYENVN__LAUNCHER__` environment variable is used to set `PyConfig.base_executable`

10.10 Py_RunMain()

int **Py_RunMain** (void)

Execute 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.

By default and when if `-i` option is used, run the REPL.

Finally, finalizes Python and returns an exit status that can be passed to the `exit()` function.

See *Python Configuration* for an example of customized Python always running in isolated mode using `Py_RunMain()`.

10.11 Py_GetArgcArgv()

void **Py_GetArgcArgv** (int **argc*, wchar_t ****argv*)

Get the original command line arguments, before Python modified them.

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.
- `PyConfig._isolated_interpreter`: if non-zero, disallow threads, subprocesses and fork.

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;

    /* ... 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);
    }
}
```

Memory Management

11.1 Overview

Memory management in Python involves a private heap containing all Python objects and data structures. The management of this private heap is ensured internally by the *Python memory manager*. The Python memory manager has different components which deal with various dynamic storage management aspects, like sharing, segmentation, preallocation or caching.

At the lowest level, a raw memory allocator ensures that there is enough room in the private heap for storing all Python-related data by interacting with the memory manager of the operating system. On top of the raw memory allocator, several object-specific allocators operate on the same heap and implement distinct memory management policies adapted to the peculiarities of every object type. For example, integer objects are managed differently within the heap than strings, tuples or dictionaries because integers imply different storage requirements and speed/space tradeoffs. The Python memory manager thus delegates some of the work to the object-specific allocators, but ensures that the latter operate within the bounds of the private heap.

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.

Ayrıca bkz.:

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 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.

3.4 sürümüyle geldi.

void* PyMem_RawMalloc (size_t *n*)

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*)

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.

3.5 sürümüyle geldi.

void* PyMem_RawRealloc (void **p*, size_t *n*)

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*)

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.

11.3 Memory Interface

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*.

Uyarı: The *GIL* must be held when using these functions.

3.6 sürümünde değişti: The default allocator is now `pymalloc` instead of `system malloc()`.

void* PyMem_Malloc (size_t *n*)

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*)

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.

3.5 sürümüyle geldi.

void* PyMem_Realloc (void **p*, size_t *n*)

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*)

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.

The following type-oriented macros are provided for convenience. Note that *TYPE* refers to any C type.

TYPE* PyMem_New (TYPE, size_t *n*)

Same as `PyMem_Malloc()`, but allocates $(n * \text{sizeof}(\text{TYPE}))$ bytes of memory. Returns a pointer cast to `TYPE*`. The memory will not have been initialized in any way.

TYPE* PyMem_Resize (void **p*, TYPE, size_t *n*)

Same as `PyMem_Realloc()`, but the memory block is resized to $(n * \text{sizeof}(\text{TYPE}))$ 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()`.

In addition, the following macro sets are provided for calling the Python memory allocator directly, without involving the C API functions listed above. However, note that their use does not preserve binary compatibility across Python versions and is therefore deprecated in extension modules.

- `PyMem_MALLOC(size)`
- `PyMem_NEW(type, size)`
- `PyMem_REALLOC(ptr, size)`
- `PyMem_RESIZE(ptr, type, size)`
- `PyMem_FREE(ptr)`
- `PyMem_DEL(ptr)`

11.4 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.

The *default object allocator* uses the *pymalloc memory allocator*.

Uyarı: The *GIL* must be held when using these functions.

`void* PyObject_Malloc(size_t n)`

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)`

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.

3.5 sürümüyle geldi.

`void* PyObject_Realloc(void *p, size_t n)`

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)`

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.

11.5 Default Memory Allocators

Default memory allocators:

Configuration	Name	Py-Mem_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*
- "+ debug": with debug hooks installed by `PyMem_SetupDebugHooks()`

11.6 Customize Memory Allocators

3.4 sürümüyle geldi.

PyMemAllocatorEx

Structure used to describe a memory block allocator. The structure has the following fields:

Field	Meaning
<code>void *ctx</code>	user context passed as first argument
<code>void* malloc(void *ctx, size_t size)</code>	allocate a memory block
<code>void* calloc(void *ctx, size_t nelem, size_t elsize)</code>	allocate a memory block initialized with zeros
<code>void* realloc(void *ctx, void *ptr, size_t new_size)</code>	allocate or resize a memory block
<code>void free(void *ctx, void *ptr)</code>	free a memory block

3.5 sürümünde değişti: The `PyMemAllocator` structure was renamed to `PyMemAllocatorEx` and a new `calloc` field was added.

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.

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.

void **PyMem_SetupDebugHooks** (void)

Setup hooks to detect bugs in the Python memory allocator functions.

Newly allocated memory is filled with the byte 0xCD (CLEANBYTE), freed memory is filled with the byte 0xDD (DEADBYTE). Memory blocks are surrounded by “forbidden bytes” (FORBIDDENBYTE: byte 0xFD).

Runtime checks:

- Detect API violations, ex: `PyObject_Free()` called on a buffer 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.

These hooks are *installed by default* if Python is compiled in debug mode. The `PYTHONMALLOC` environment variable can be used to install debug hooks on a Python compiled in release mode.

3.6 sürümünde değişti: This 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.

3.8 sürümünde değişti: Byte patterns 0xCB (CLEANBYTE), 0xDB (DEADBYTE) and 0xFB (FORBIDDENBYTE) have been replaced with 0xCD, 0xDD and 0xFD to use the same values than Windows CRT debug `malloc()` and `free()`.

11.7 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 256 KiB. 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.

11.7.1 Customize pymalloc Arena Allocator

3.4 sürümüyle geldi.

PyObjectArenaAllocator

Structure used to describe an arena allocator. The structure has three fields:

Field	Meaning
<code>void *ctx</code>	user context passed as first argument
<code>void* alloc(void *ctx, size_t size)</code>	allocate an arena of size bytes
<code>void free(void *ctx, void *ptr, size_t size)</code>	free an arena

void **PyObject_GetArenaAllocator** (*PyObjectArenaAllocator *allocator*)
Get the arena allocator.

void **PyObject_SetArenaAllocator** (*PyObjectArenaAllocator *allocator*)
Set the arena allocator.

11.8 tracemalloc C API

3.7 sürümüyle geldi.

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.

11.9 Examples

Here is the example from section *Overview*, rewritten so that the I/O buffer is allocated from the Python heap by using the first function set:

```
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;
```

The same code using the type-oriented function set:

```
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;
```

Note that in the two examples above, the buffer is always manipulated via functions belonging to the same set. Indeed, it is required to use the same memory API family for a given memory block, so that the risk of mixing different allocators is reduced to a minimum. The following code sequence contains two errors, one of which is labeled as *fatal* because it mixes two different allocators operating on different heaps.

```
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()`.

These will be explained in the next chapter on defining and implementing new object types in C.

Object Implementation Support

This chapter describes the functions, types, and macros used when defining new object types.

12.1 Allocating Objects on the Heap

*PyObject** **_PyObject_New** (*PyTypeObject* *type)

Return value: New reference.

*PyVarObject** **_PyObject_NewVar** (*PyTypeObject* *type, *Py_ssize_t* size)

Return value: New reference.

*PyObject** **PyObject_Init** (*PyObject* *op, *PyTypeObject* *type)

Return value: Borrowed reference. Initialize a newly-allocated object *op* with its type and initial reference. Returns the initialized object. If *type* indicates that the object participates in the cyclic garbage detector, it is added to the detector's set of observed objects. Other fields of the object are not affected.

*PyVarObject** **PyObject_InitVar** (*PyVarObject* *op, *PyTypeObject* *type, *Py_ssize_t* size)

Return value: Borrowed reference. This does everything *PyObject_Init()* does, and also initializes the length information for a variable-size object.

*TYPE** **PyObject_New** (*TYPE*, *PyTypeObject* *type)

Return value: New reference. Allocate a new Python object using the C structure type *TYPE* and the Python type object *type*. Fields not defined by the Python object header are not initialized; the object's reference count will be one. The size of the memory allocation is determined from the *tp_basicsize* field of the type object.

*TYPE** **PyObject_NewVar** (*TYPE*, *PyTypeObject* *type, *Py_ssize_t* size)

Return value: New reference. Allocate a new Python object using the C structure type *TYPE* and the Python type object *type*. Fields not defined by the Python object header are not initialized. The allocated memory allows for the *TYPE* structure plus *size* fields of the size given by the *tp_itemsize* field of *type*. 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**

Object which is visible in Python as `None`. This should only be accessed using the *Py_None* macro, which evaluates to a pointer to this object.

Ayrıca bkz.:

PyModule_Create() To allocate and create extension modules.

12.2 Common Object Structures

There are a large number of structures which are used in the definition of object types for Python. This section describes these structures and how they are used.

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.

PyObject

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*.

PyVarObject

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.

***Py_TYPE* (o)**

This macro is used to access the `ob_type` member of a Python object. It expands to:

```
((PyObject*) (o)) ->ob_type)
```

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`.

3.9 sürümüyle geldi.

void ***Py_SET_TYPE*** (*PyObject* *o, *PyTypeObject* *type)

Set the object *o* type to *type*.

3.9 sürümüyle geldi.

Py_REFCNT (o)

This macro is used to access the `ob_refcnt` member of a Python object. It expands to:

```
((PyObject*) (o))->ob_refcnt
```

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

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

3.9 sürümüyle geldi.

Py_SIZE (o)

This macro is used to access the `ob_size` member of a Python object. It expands to:

```
((PyVarObject*) (o))->ob_size
```

void **Py_SET_SIZE** (*PyVarObject* *o, *Py_ssize_t* size)

Set the object *o* size to *size*.

3.9 sürümüyle geldi.

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

PyCFunction

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);
```

PyCFunctionWithKeywords

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);
```

_PyCFunctionFast

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);
```

`_PyCFunctionFastWithKeywords`

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);
```

`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)
```

3.9 sürümüyle geldi.

`PyMethodDef`

Structure used to describe a method of an extension type. This structure has four fields:

Field	C Type	Meaning
<code>ml_name</code>	<code>const char *</code>	name of the method
<code>ml_meth</code>	<code>PyCFunction</code>	pointer to the C implementation
<code>ml_flags</code>	<code>int</code>	flag bits indicating how the call should be constructed
<code>ml_doc</code>	<code>const char *</code>	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_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).

This is not part of the *limited API*.

3.7 sürümüyle geldi.

`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.

This is not part of the *limited API*.

3.7 sürümüyle geldi.

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`.

3.9 sürümüyle geldi.

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`.

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.

These two constants are not used to indicate the calling convention but the binding when use with methods of classes. These may not be used for functions defined for modules. At most one of these flags may be set for any given method.

METH_CLASS

The method will be passed the type object as the first parameter rather than an instance of the type. This is used to create *class methods*, similar to what is created when using the `classmethod()` built-in function.

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.

One other constant controls whether a method is loaded in place of another definition with the same method name.

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.

12.2.3 Accessing attributes of extension types

PyMemberDef

Structure which describes an attribute of a type which corresponds to a C struct member. Its fields are:

Field	C Type	Meaning
<code>name</code>	<code>const char *</code>	name of the member
<code>type</code>	<code>int</code>	the type of the member in the C struct
<code>offset</code>	<code>Py_ssize_t</code>	the offset in bytes that the member is located on the type's object struct
<code>flags</code>	<code>int</code>	flag bits indicating if the field should be read-only or writable
<code>doc</code>	<code>const char *</code>	points to the contents of the docstring

`type` can be one of many `T_` macros corresponding to various C types. When the member is accessed in Python, it will be converted to the equivalent Python type.

Macro name	C type
<code>T_SHORT</code>	short
<code>T_INT</code>	int
<code>T_LONG</code>	long
<code>T_FLOAT</code>	float
<code>T_DOUBLE</code>	double
<code>T_STRING</code>	const char *
<code>T_OBJECT</code>	PyObject *
<code>T_OBJECT_EX</code>	PyObject *
<code>T_CHAR</code>	char
<code>T_BYTE</code>	char
<code>T_UBYTE</code>	unsigned char
<code>T_UINT</code>	unsigned int
<code>T_USHORT</code>	unsigned short
<code>T_ULONG</code>	unsigned long
<code>T_BOOL</code>	char
<code>T_LONGLONG</code>	long long
<code>T_ULONGLONG</code>	unsigned long long
<code>T_PYSSIZET</code>	Py_ssize_t

`T_OBJECT` and `T_OBJECT_EX` differ in that `T_OBJECT` returns `None` if the member is `NULL` and `T_OBJECT_EX` raises an `AttributeError`. Try to use `T_OBJECT_EX` over `T_OBJECT` because `T_OBJECT_EX` handles use of the `del` statement on that attribute more correctly than `T_OBJECT`.

`flags` can be 0 for write and read access or `READONLY` for read-only access. Using `T_STRING` for `type` implies `READONLY`. `T_STRING` data is interpreted as UTF-8. Only `T_OBJECT` and `T_OBJECT_EX` members can be deleted. (They are set to `NULL`).

Heap allocated types (created using `PyType_FromSpec()` or similar), `PyMemberDef` may contain definitions for the special members `__dictoffset__`, `__weaklistoffset__` and `__vectorcalloffset__`, corresponding to `tp_dictoffset`, `tp_weaklistoffset` and `tp_vectorcall_offset` in type objects. These must be defined with `T_PYSSIZET` and `READONLY`, for example:

```
static PyMemberDef spam_type_members[] = {
    {"__dictoffset__", T_PYSSIZET, offsetof(Spam_object, dict), READONLY},
    {NULL} /* Sentinel */
};
```

*PyObject** **PyMember_GetOne** (const char **obj_addr*, struct *PyMemberDef* **m*)

Get an attribute belonging to the object at address *obj_addr*. The attribute is described by `PyMemberDef` *m*. Returns `NULL` on error.

int **PyMember_SetOne** (char **obj_addr*, struct *PyMemberDef* **m*, *PyObject* **o*)

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.

PyGetSetDef

Structure to define property-like access for a type. See also description of the `PyTypeObject.tp_getset` slot.

Field	C Type	Meaning
name	const char *	attribute name
get	getter	C function to get the attribute
set	setter	optional C function to set or delete the attribute, if omitted the attribute is readonly
doc	const char *	optional docstring
closure	void *	optional function pointer, providing additional data for getter and setter

The `get` function takes one *PyObject** parameter (the instance) and a function pointer (the associated closure):

```
typedef PyObject *(*getter)(PyObject *, void *);
```

It should return a new reference on success or NULL with a set exception on failure.

set functions take two *PyObject** parameters (the instance and the value to be set) and a function pointer (the associated closure):

```
typedef int (*setter)(PyObject *, PyObject *, void *);
```

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 *Examples* section provides at-a-glance insight into the meaning and use of *PyTypeObject*.

12.3.1 Quick Reference

“tp slots”

PyTypeObject Slot ¹	Type	special methods/attrs	Info ²			
			O	T	D	I
<R> <i>tp_name</i>	const char *	<i>__name__</i>	X	X		
<i>tp_basicsize</i>	<i>Py_ssize_t</i>		X	X		X
<i>tp_itemsize</i>	<i>Py_ssize_t</i>			X		X
<i>tp_dealloc</i>	destructor		X	X		X
<i>tp_vectorcall_offset</i>	<i>Py_ssize_t</i>			X		X
(<i>tp_getattr</i>)	<i>getattrfunc</i>	<i>__getattribute__</i> , <i>__getattr__</i>				G
(<i>tp_setattr</i>)	<i>setattrfunc</i>	<i>__setattr__</i> , <i>__delattr__</i>				G
<i>tp_as_async</i>	<i>PyAsyncMethods</i> *	<i>sub-slots</i>				%
<i>tp_repr</i>	<i>reprfunc</i>	<i>__repr__</i>	X	X		X

Devamı sonraki sayfada

Tablo 1 – önceki sayfadan devam

PyTypeObject Slot ¹	Type	special methods/attrs	Info ²				
			O	T	D	I	
<code>tp_as_number</code>	<code>PyNumberMethods *</code>	<code>sub-slots</code>					%
<code>tp_as_sequence</code>	<code>PySequenceMethods *</code>	<code>sub-slots</code>					%
<code>tp_as_mapping</code>	<code>PyMappingMethods *</code>	<code>sub-slots</code>					%
<code>tp_hash</code>	<code>hashfunc</code>	<code>__hash__</code>	X				G
<code>tp_call</code>	<code>ternaryfunc</code>	<code>__call__</code>		X			X
<code>tp_str</code>	<code>reprfunc</code>	<code>__str__</code>	X				X
<code>tp_getattro</code>	<code>getattrofunc</code>	<code>__getattr__</code> , <code>__getattribute__</code>	X	X			G
<code>tp_setattro</code>	<code>setattrofunc</code>	<code>__setattr__</code> , <code>__delattr__</code>	X	X			G
<code>tp_as_buffer</code>	<code>PyBufferProcs *</code>						%
<code>tp_flags</code>	unsigned long		X	X			?
<code>tp_doc</code>	const char *	<code>__doc__</code>	X	X			
<code>tp_traverse</code>	<code>traverseproc</code>			X			G
<code>tp_clear</code>	<code>inquiry</code>			X			G
<code>tp_richcompare</code>	<code>richcmpfunc</code>	<code>__lt__</code> , <code>__le__</code> , <code>__eq__</code> , <code>__ne__</code> , <code>__gt__</code> , <code>__ge__</code>	X				G
<code>tp_weaklistoffset</code>	<code>Py_ssize_t</code>			X			?
<code>tp_iter</code>	<code>getiterfunc</code>	<code>__iter__</code>					X
<code>tp_iternext</code>	<code>iternextfunc</code>	<code>__next__</code>					X
<code>tp_methods</code>	<code>PyMethodDef []</code>		X	X			
<code>tp_members</code>	<code>PyMemberDef []</code>			X			
<code>tp_getset</code>	<code>PyGetSetDef []</code>		X	X			
<code>tp_base</code>	<code>PyTypeObject *</code>	<code>__base__</code>				X	
<code>tp_dict</code>	<code>PyObject *</code>	<code>__dict__</code>				?	
<code>tp_descr_get</code>	<code>descrgetfunc</code>	<code>__get__</code>					X
<code>tp_descr_set</code>	<code>descrsetfunc</code>	<code>__set__</code> , <code>__delete__</code>					X
<code>tp_dictoffset</code>	<code>Py_ssize_t</code>			X			?
<code>tp_init</code>	<code>initproc</code>	<code>__init__</code>	X	X			X
<code>tp_alloc</code>	<code>allocfunc</code>		X		?	?	
<code>tp_new</code>	<code>newfunc</code>	<code>__new__</code>	X	X	?	?	
<code>tp_free</code>	<code>freefunc</code>		X	X	?	?	
<code>tp_is_gc</code>	<code>inquiry</code>			X			X
<code><tp_bases></code>	<code>PyObject *</code>	<code>__bases__</code>				~	
<code><tp_mro></code>	<code>PyObject *</code>	<code>__mro__</code>				~	
<code>[tp_cache]</code>	<code>PyObject *</code>						
<code>[tp_subclasses]</code>	<code>PyObject *</code>	<code>__subclasses__</code>					
<code>[tp_weaklist]</code>	<code>PyObject *</code>						
<code>(tp_del)</code>	<code>destructor</code>						
<code>[tp_version_tag]</code>	unsigned int						
<code>tp_finalize</code>	<code>destructor</code>	<code>__del__</code>					X
<code>tp_vectorcall</code>	<code>vectorcallfunc</code>						

¹ A slot name in parentheses indicates it is (effectively) deprecated. Names in angle brackets should be 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

sub-slots

Slot	Type	special methods
<i>am_await</i>	<i>unaryfunc</i>	<i>__await__</i>
<i>am_aiter</i>	<i>unaryfunc</i>	<i>__aiter__</i>
<i>am_anext</i>	<i>unaryfunc</i>	<i>__anext__</i>
<i>nb_add</i>	<i>binaryfunc</i>	<i>__add__</i> <i>__radd__</i>
<i>nb_inplace_add</i>	<i>binaryfunc</i>	<i>__iadd__</i>
<i>nb_subtract</i>	<i>binaryfunc</i>	<i>__sub__</i> <i>__rsub__</i>
<i>nb_inplace_subtract</i>	<i>binaryfunc</i>	<i>__isub__</i>
<i>nb_multiply</i>	<i>binaryfunc</i>	<i>__mul__</i> <i>__rmul__</i>
<i>nb_inplace_multiply</i>	<i>binaryfunc</i>	<i>__imul__</i>
<i>nb_remainder</i>	<i>binaryfunc</i>	<i>__mod__</i> <i>__rmod__</i>
<i>nb_inplace_remainder</i>	<i>binaryfunc</i>	<i>__imod__</i>
<i>nb_divmod</i>	<i>binaryfunc</i>	<i>__divmod__</i> <i>__rdivmod__</i>
<i>nb_power</i>	<i>ternaryfunc</i>	<i>__pow__</i> <i>__rpow__</i>
<i>nb_inplace_power</i>	<i>ternaryfunc</i>	<i>__ipow__</i>
<i>nb_negative</i>	<i>unaryfunc</i>	<i>__neg__</i>
<i>nb_positive</i>	<i>unaryfunc</i>	<i>__pos__</i>
<i>nb_absolute</i>	<i>unaryfunc</i>	<i>__abs__</i>
<i>nb_bool</i>	<i>inquiry</i>	<i>__bool__</i>
<i>nb_invert</i>	<i>unaryfunc</i>	<i>__invert__</i>
<i>nb_lshift</i>	<i>binaryfunc</i>	<i>__lshift__</i> <i>__rlshift__</i>
<i>nb_inplace_lshift</i>	<i>binaryfunc</i>	<i>__ilshift__</i>
<i>nb_rshift</i>	<i>binaryfunc</i>	<i>__rshift__</i> <i>__rrshift__</i>
<i>nb_inplace_rshift</i>	<i>binaryfunc</i>	<i>__irshift__</i>
<i>nb_and</i>	<i>binaryfunc</i>	<i>__and__</i> <i>__rand__</i>
<i>nb_inplace_and</i>	<i>binaryfunc</i>	<i>__iand__</i>
<i>nb_xor</i>	<i>binaryfunc</i>	<i>__xor__</i> <i>__rxor__</i>
<i>nb_inplace_xor</i>	<i>binaryfunc</i>	<i>__ixor__</i>
<i>nb_or</i>	<i>binaryfunc</i>	<i>__or__</i> <i>__ror__</i>
<i>nb_inplace_or</i>	<i>binaryfunc</i>	<i>__ior__</i>
<i>nb_int</i>	<i>unaryfunc</i>	<i>__int__</i>
<i>nb_reserved</i>	void *	
<i>nb_float</i>	<i>unaryfunc</i>	<i>__float__</i>
<i>nb_floor_divide</i>	<i>binaryfunc</i>	<i>__floordiv__</i>
<i>nb_inplace_floor_divide</i>	<i>binaryfunc</i>	<i>__ifloordiv__</i>
<i>nb_true_divide</i>	<i>binaryfunc</i>	<i>__truediv__</i>
<i>nb_inplace_true_divide</i>	<i>binaryfunc</i>	<i>__itruediv__</i>
<i>nb_index</i>	<i>unaryfunc</i>	<i>__index__</i>

Devamı sonraki sayfada

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.

Tablo 2 – önceki sayfadan devam

Slot	Type	special methods
<i>nb_matrix_multiply</i>	<i>binaryfunc</i>	<i>__matmul__ __rmatmul__</i>
<i>nb_inplace_matrix_multiply</i>	<i>binaryfunc</i>	<i>__imatmul__</i>
<i>mp_length</i>	<i>lenfunc</i>	<i>__len__</i>
<i>mp_subscript</i>	<i>binaryfunc</i>	<i>__getitem__</i>
<i>mp_ass_subscript</i>	<i>objobjargproc</i>	<i>__setitem__, __delitem__</i>
<i>sq_length</i>	<i>lenfunc</i>	<i>__len__</i>
<i>sq_concat</i>	<i>binaryfunc</i>	<i>__add__</i>
<i>sq_repeat</i>	<i>ssizeargfunc</i>	<i>__mul__</i>
<i>sq_item</i>	<i>ssizeargfunc</i>	<i>__getitem__</i>
<i>sq_ass_item</i>	<i>ssizeobjargproc</i>	<i>__setitem__, __delitem__</i>
<i>sq_contains</i>	<i>objobjproc</i>	<i>__contains__</i>
<i>sq_inplace_concat</i>	<i>binaryfunc</i>	<i>__iadd__</i>
<i>sq_inplace_repeat</i>	<i>ssizeargfunc</i>	<i>__imul__</i>
<i>bf_getbuffer</i>	<i>getbufferproc()</i>	
<i>bf_releasebuffer</i>	<i>releasebufferproc()</i>	

slot typedefs

typedef	Parameter Types	Return Type
<i>allocfunc</i>	<i>PyTypeObject</i> * <i>Py_ssize_t</i>	<i>PyObject</i> *
<i>destructor</i>	void *	void
<i>freefunc</i>	void *	void
<i>traverseproc</i>	void * <i>visitproc</i> void *	int
<i>newfunc</i>	<i>PyObject</i> * <i>PyObject</i> * <i>PyObject</i> *	<i>PyObject</i> *
<i>initproc</i>	<i>PyObject</i> * <i>PyObject</i> * <i>PyObject</i> *	int
<i>reprfunc</i>	<i>PyObject</i> *	<i>PyObject</i> *
<i>getattrfunc</i>	<i>PyObject</i> * const char *	<i>PyObject</i> *
<i>setattrfunc</i>	<i>PyObject</i> * const char * <i>PyObject</i> *	int
<i>getattrofunc</i>	<i>PyObject</i> * <i>PyObject</i> *	<i>PyObject</i> *
<i>setattrofunc</i>	<i>PyObject</i> * <i>PyObject</i> * <i>PyObject</i> *	int
<i>descrgetfunc</i>	<i>PyObject</i> * <i>PyObject</i> * <i>PyObject</i> *	<i>PyObject</i> *
<i>descrsetfunc</i>	<i>PyObject</i> * <i>PyObject</i> *	int
12.3. Type Objects	<i>PyObject</i> *	193
<i>hashfunc</i>	<i>PyObject</i> *	<i>Py_hash_t</i>
<i>richcmpfunc</i>		<i>PyObject</i> *

See *Slot Type typedefs* below for more detail.

12.3.2 PyObject Definition

The structure definition for *PyObject* 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;
    getattrofunc tp_getattr;
    setattrofunc 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 */

    /* call function for all accessible objects */
    traverseproc tp_traverse;

    /* delete references to contained objects */
    inquiry tp_clear;

    /* rich comparisons */
    richcmpfunc tp_richcompare;

    /* weak reference enabler */
    Py_ssize_t tp_weaklistoffset;

    /* Iterators */
    getiterfunc tp_iter;
    iternextfunc tp_iternext;

    /* Attribute descriptor and subclassing stuff */
```

(continues on next page)

(önceki sayfadan devam)

```

struct PyMethodDef *tp_methods;
struct PyMemberDef *tp_members;
struct PyGetSetDef *tp_getset;
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;
} 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.

*PyObject** **PyObject._ob_next**

*PyObject** **PyObject._ob_prev**

These fields are only present when the macro `Py_TRACE_REFS` is defined. Their initialization to `NULL` is taken care of by the `PyObject_HEAD_INIT` macro. For statically allocated objects, these fields always remain `NULL`. For dynamically allocated objects, these two fields are used to link the object into a doubly-linked list of *all* live objects on the heap. This could be used for various debugging purposes; currently the only use is to print the objects that are still alive at the end of a run when the environment variable `PYTHONDUMPREFS` is set.

Inheritance:

These fields are not inherited by subtypes.

Py_ssize_t **PyObject.ob_refcnt**

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**

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**

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_VarNew()`, 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 decrement the reference count for its type object 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 **PyObject.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()*.

Uyarı: It is not recommended for *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.

Not: The semantics of the *tp_vectorcall_offset* slot are provisional and expected to be finalized in Python 3.9. If you use vectorcall, plan for updating your code for Python 3.9.

3.8 sürümünde değişti: 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.

Inheritance:

This field is always inherited. However, the *Py_TPFLAGS_HAVE_VECTORCALL* flag is not always inherited. If it's not, then the subclass won't use *vectorcall*, except when *PyVectorcall_Call()* is explicitly called. This is in particular the case for *heap types* (including subclasses defined in Python).

getattrfunc **PyObject.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 **PyObject.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** **PyObject.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.

3.5 sürümüyle geldi: 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 **PyObject.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 `<object at %p>` is returned, where `%s` is replaced by the type name, and `%p` by the object's memory address.

*PyNumberMethods** **PyObject.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** **PyObject.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** **PyObject.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 **PyObject.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.

ternaryfunc **PyObject.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 **PyObject.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.

getattrfunc **PyObject.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()`.

setattrfunc **PyObject.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.

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).

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`, `Py_TPFLAGS_HAVE_VERSION_TAG`.

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`.

3.8 sürümüyle geldi.

Inheritance:

This flag is never inherited by heap types. For extension types, it is inherited whenever `tp_descr_get` 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.

3.4 sürümüyle geldi.

3.8 sürümünden beri önerilmiyor: 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 for *static* subtypes if `tp_call` is also inherited. *Heap types* do not inherit `Py_TPFLAGS_HAVE_VECTORCALL`.

3.9 sürümüyle geldi.

`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.

Uyarı: 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.

Heap-allocated types (`Py_TPFLAGS_HEAPTYPE`, such as those created with `PyType_FromSpec()` and similar APIs) 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.

3.9 sürümünde değişti: 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 decremented until after the pointer to the contained object is set to `NULL`. This is because decrementing the reference count 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.

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 **PyObject.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 `PyObject`.

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()`:

Constant	Comparison
<code>Py_LT</code>	<code><</code>
<code>Py_LE</code>	<code>< =</code>
<code>Py_EQ</code>	<code>==</code>
<code>Py_NE</code>	<code>!=</code>
<code>Py_GT</code>	<code>></code>
<code>Py_GE</code>	<code>> =</code>

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 return value's reference count is properly incremented.

On error, sets an exception and returns `NULL` from the function.

3.7 sürümüyle geldi.

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 **PyObject.tp_weaklistoffset**

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.

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.

When a type defined by a class statement has no `__slots__` declaration, and none of its base types are weakly referenceable, the type is made weakly referenceable by adding a weak reference list head slot to the instance layout and setting the `tp_weaklistoffset` of that slot's offset.

When a type's `__slots__` declaration contains a slot named `__weakref__`, that slot becomes the weak reference list head for instances of the type, and the slot's offset is stored in the type's `tp_weaklistoffset`.

When a type's `__slots__` declaration does not contain a slot named `__weakref__`, the type inherits its `tp_weaklistoffset` from its base type.

getterfunc **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.

Not: 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__()*).

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.

Uyarı: 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`

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.

If the value of this field is greater than zero, it specifies the offset from the start of the instance structure. If the value is less than zero, it specifies the offset from the *end* of the instance structure. A negative offset is more expensive to use, and should only be used when the instance structure contains a variable-length part. This is used for example to add an instance variable dictionary to subtypes of `str` or `tuple`. Note that the `tp_basicsize` field should account for the dictionary added to the end in that case, even though the dictionary is not included in the basic object layout. On a system with a pointer size of 4 bytes, `tp_dictoffset` should be set to `-4` to indicate that the dictionary is at the very end of the structure.

The real dictionary offset in an instance can be computed from a negative `tp_dictoffset` as follows:

```
dictoffset = tp_basicsize + abs(ob_size)*tp_itemsize + tp_dictoffset
if dictoffset is not aligned on sizeof(void*):
    round up to sizeof(void*)
```

where `tp_basicsize`, `tp_itemsize` and `tp_dictoffset` are taken from the type object, and `ob_size` is taken from the instance. The absolute value is taken because ints use the sign of `ob_size` to store the sign of the number. (There's never a need to do this calculation yourself; it is done for you by `_PyObject_GetDictPtr()`.)

Inheritance:

This field is inherited by subtypes, but see the rules listed below. A subtype may override this offset; this means that the subtype instances store the dictionary at a difference offset than the base type. Since the dictionary is always found via `tp_dictoffset`, this should not be a problem.

When a type defined by a class statement has no `__slots__` declaration, and none of its base types has an instance variable dictionary, a dictionary slot is added to the instance layout and the `tp_dictoffset` is set to that slot's offset.

When a type defined by a class statement has a `__slots__` declaration, the type inherits its `tp_dictoffset` from its base type.

(Adding a slot named `__dict__` to the `__slots__` declaration does not have the expected effect, it just causes confusion. Maybe this should be added as a feature just like `__weakref__` though.)

Default:

This slot has no default. For static types, if the field is `NULL` then no `__dict__` gets created for instances.

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 `kwargs` 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 *kwargs);
```

The `subtype` argument is the type of the object being created; the `args` and `kwargs` 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`.

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 is set for types created by a class statement. It should be `NULL` for statically defined types.

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.

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.

*PyObject** **`PyTypeObject.tp_subclasses`**

List of weak references to subclasses. Internal use only.

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.

Inheritance:

This field is not inherited.

destructor **PyObject.tp_del**

This field is deprecated. Use *tp_finalize* instead.

unsigned int **PyObject.tp_version_tag**

Used to index into the method cache. Internal use only.

Inheritance:

This field is not inherited.

destructor **PyObject.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);
}
```

For this field to be taken into account (even through inheritance), you must also set the *Py_TPFLAGS_HAVE_FINALIZE* flags bit.

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.

3.4 sürümüyle geldi.

Ayrıca bkz.:

“Safe object finalization” (**PEP 442**)

vectorcallfunc **PyObject.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.

3.9 sürümüyle geldi: (the field exists since 3.8 but it's only used since 3.9)

12.3.6 Heap 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 not part of the *stable ABI*, any extension modules using static types must be compiled for a specific Python minor version.

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.

This is done by filling a `PyType_Spec` structure and calling `PyType_FromSpecWithBases()`.

12.4 Number Object Structures

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;
    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;
```

(continues on next page)

(önceki sayfadan devam)

```

    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;

```

Not: 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.

Not: 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

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()*, *PyObject_SetSlice()* and *PyObject_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

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

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 `PyExc_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

3.5 sürümüyle geldi.

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;
} 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.

12.9 Slot Type typedefs

PyObject * (***allocfunc**) (*PyTypeObject* *cls, *Py_ssize_t* nitems)

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`.

void (***destructor**) (*PyObject* *)

void (***freefunc**) (void *)

See `tp_free`.

PyObject * (***newfunc**) (*PyObject* *, *PyObject* *, *PyObject* *)

See `tp_new`.

int (***inittest**) (*PyObject* *, *PyObject* *, *PyObject* *)

See `tp_init`.

PyObject * (***reprfunc**) (*PyObject* *)

See `tp_repr`.

PyObject * (***getattrfunc**) (*PyObject* *self, char *attr)

Return the value of the named attribute for the object.

int (***setattrfunc**) (*PyObject* *self, char *attr, *PyObject* *value)

Set the value of the named attribute for the object. The value argument is set to NULL to delete the attribute.

PyObject * (***getattrofunc**) (*PyObject* *self, *PyObject* *attr)

Return the value of the named attribute for the object.

See `tp_getattro`.

int (***setattrofunc**) (*PyObject* *self, *PyObject* *attr, *PyObject* *value)

Set the value of the named attribute for the object. The value argument is set to NULL to delete the attribute.

See `tp_setattro`.

PyObject * (***descrgetfunc**) (*PyObject* *, *PyObject* *, *PyObject* *)

See `tp_descrget`.

int (***descrsetfunc**) (*PyObject* *, *PyObject* *, *PyObject* *)

See `tp_descrset`.

Py_hash_t (***hashfunc**) (*PyObject* *)

See `tp_hash`.

PyObject * (***richcmpfunc**) (*PyObject* *, *PyObject* *, int)

See `tp_richcompare`.

```
PyObject* (*getiterfunc) (PyObject *)
    See tp_iter.

PyObject* (*iternextfunc) (PyObject *)
    See tp_iternext.

Py_ssize_t (*lenfunc) (PyObject *)

int (*getbufferproc) (PyObject *, Py_buffer *, int)

void (*releasebufferproc) (PyObject *, Py_buffer *)

PyObject* (*unaryfunc) (PyObject *)

PyObject* (*binaryfunc) (PyObject *, PyObject *)

PyObject* (*ternaryfunc) (PyObject *, PyObject *, PyObject *)

PyObject* (*ssizeargfunc) (PyObject *, Py_ssize_t)

int (*ssizeobjargproc) (PyObject *, Py_ssize_t)

int (*objobjproc) (PyObject *, PyObject *)

int (*objobjargproc) (PyObject *, PyObject *, PyObject *)
```

12.10 Examples

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;
} PyObject;

static PyTypeObject PyObject_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 PyObject_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 */
```

(continues on next page)

(önceki sayfadan devam)

```

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;
    PyObject *inst_dict;
    PyObject *weakreflist;
} PyObject;

static PyTypeObject PyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyObject",
    .tp_basicsize = sizeof(PyObject),
    .tp_doc = PyDoc_STR("My objects"),
    .tp_weaklistoffset = offsetof(PyObject, weakreflist),
    .tp_dictoffset = offsetof(PyObject, inst_dict),
    .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE | Py_TPFLAGS_HAVE_GC,
    .tp_new = myobj_new,
    .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):

```

typedef struct {
    PyUnicodeObject raw;
    char *extra;
} MyStr;

```

(continues on next page)

```
static PyObject 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,
    .tp_new = NULL,
    .tp_repr = (reprfunc)myobj_repr,
};
```

The simplest static type (with fixed-length instances):

```
typedef struct {
    PyObject_HEAD
} MyObject;

static PyObject 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 PyObject 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()`.

Uyarı: 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.

TYPE* `PyObject_GC_New` (TYPE, *PyTypeObject* *type)

Analogous to `PyObject_New()` but for container objects with the `Py_TPFLAGS_HAVE_GC` flag set.

TYPE* `PyObject_GC_NewVar` (TYPE, *PyTypeObject* *type, *Py_ssize_t* size)

Analogous to `PyObject_NewVar()` but for container objects with the `Py_TPFLAGS_HAVE_GC` flag set.

TYPE* `PyObject_GC_Resize` (TYPE, *PyVarObject* *op, *Py_ssize_t* newsize)

Resize an object allocated by `PyObject_NewVar()`. Returns the resized object or NULL on failure. *op* must not be tracked by the collector yet.

void `PyObject_GC_Track` (*PyObject* *op)

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)

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()`.

3.9 sürümüyle geldi.

int `PyObject_GC_IsFinalized` (*PyObject* *op)

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()`.

3.9 sürümüyle geldi.

void `PyObject_GC_Del` (void *op)

Releases memory allocated to an object using `PyObject_GC_New()` or `PyObject_GC_NewVar()`.

void `PyObject_GC_UnTrack` (void *op)

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.

3.8 sürümünde değişti: 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:

int (***visitproc**) (*PyObject* *object, void *arg)

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:

`int (*traverseproc) (PyObject *self, visitproc visit, void *arg)`

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.

`int (*inquiry) (PyObject *self)`

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.

API and ABI Versioning

PY_VERSION_HEX is the Python version number encoded in a single integer.

For example if the PY_VERSION_HEX is set to 0x030401a2, the underlying version information can be found by treating it as a 32 bit number in the following manner:

Bytes	Bits (big endian order)	Meaning
1	1-8	PY_MAJOR_VERSION (the 3 in 3.4.1a2)
2	9-16	PY_MINOR_VERSION (the 4 in 3.4.1a2)
3	17-24	PY_MICRO_VERSION (the 1 in 3.4.1a2)
4	25-28	PY_RELEASE_LEVEL (0xA for alpha, 0xB for beta, 0xC for release candidate and 0xF for final), in this case it is alpha.
	29-32	PY_RELEASE_SERIAL (the 2 in 3.4.1a2, zero for final releases)

Thus 3.4.1a2 is hexversion 0x030401a2.

All the given macros are defined in [Include/patchlevel.h](#).

>>> The default Python prompt of the interactive shell. Often seen for code examples which can be executed interactively in the interpreter.

... Şunlara başvurulabilir:

- Girintili bir kod bloğu için kod girerken, eşleşen bir çift sol ve sağ sınırlayıcı (parantez, köşeli parantez, kaşlı ayraç veya üçlü tırnak) içindeyken veya bir dekoratör belirttikten sonra etkileşimli kabuğun varsayılan Python istemi.
- Elipsis yerleşik sabiti.

2to3 Kaynağı ayrıştırarak ve ayrıştırma ağacında gezinerek tespit edilebilecek uyumsuzlukların çoğunu işleyerek Python 2.x kodunu Python 3.x koduna dönüştürmeye çalışan bir araç.

2to3, standart kitaplıkta `lib2to3`; bağımsız bir giriş noktası şu şekilde sağlanır: `file:Tools/scripts/2to3`. Bakınız [2to3-reference](#).

soyut temel sınıf Soyut temel sınıflar *duck-typing* 'i, `hasattr()` gibi diğer teknikler beceriksiz veya tamamen yanlış olduğunda arayüzleri tanımlamanın bir yolunu sağlayarak tamamlar (örneğin sihirli yöntemlerle). ABC'ler, bir sınıftan miras almayan ancak yine de `isinstance()` ve `issubclass()` tarafından tanınan sınıflar olan sanal alt sınıfları tanıtır; `abc` modül belgelerine bakın. 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). `abc` modülü ile kendi ABC'lerinizi oluşturabilirsiniz.

dipnot A label associated with a variable, a class attribute or a function parameter or return value, used by convention as a *type hint*.

Yerel değişkenlerin açıklamalarına çalışma zamanında erişilemez, ancak global değişkenlerin, sınıf niteliklerinin ve işlevlerin açıklamaları, sırasıyla modüllerin, sınıfların ve işlevlerin `__annotations__` özel özelliğinde saklanır.

See [variable annotation](#), [function annotation](#), [PEP 484](#) and [PEP 526](#), which describe this functionality.

argüman A value passed to a *function* (or *method*) when calling the function. There are two kinds of argument:

- *keyword argument*: bir işlev çağrısında bir tanımlayıcının (ör. `ad =`) önüne geçen veya bir sözlükte `**` ile başlayan bir değer olarak geçirilen bir argüman. Örneğin, 3 ve 5, aşağıdaki `complex()`: çağrılarında anahtar kelimenin argümanleridir:

```
complex(real=3, imag=5)
complex(**{'real': 3, 'imag': 5})
```


- *positional argument*: anahtar kelime argümanı olmayan bir argüman. Konumsal argümanlar, bir argüman listesinin başında görünebilir ve/veya * ile başlayan bir *iterable* öğesinin öğeleri olarak iletilir. Örneğin, 3 ve 5, aşağıdaki çağrılarda konumsal argümanlardır:

```
complex(3, 5)
complex(*(3, 5))
```

argümanlar, bir işlev gövdesindeki adlandırılmış yerel değişkenlere atanır. Bu atamayı yöneten kurallar için `calls` bölümüne bakın. Sözdizimsel olarak, bir argümanı temsil etmek için herhangi bir ifade kullanılabilir; değerlendirilen değer yerel değişkene atanır.

See also the *parameter* glossary entry, the FAQ question on the difference between arguments and parameters, and **PEP 362**.

asenkron bağlam yöneticisi `async with` ifadesinde görülen ortamı `__aenter__()` ve `__aexit__()` yöntemlerini tanımlayarak kontrol eden bir nesne. **PEP 492** de anlatıldı.

asenkron jeneratör *asynchronous generator iterator* döndüren bir işlev. Bir `async for` döngüsünde kullanılabilen bir dizi değer üretmek için `yield` ifadeleri içermesi dışında `async def` ile tanımlanmış bir eşyordam işlevine benziyor.

Genellikle bir asenkron üretici işlevine atıfta bulunur, ancak bazı bağlamlarda bir *asynchronous generator iterator* 'e karşılık gelebilir. Amaçlanan anlamın net olmadığı durumlarda, tam terimlerin kullanılması belirsizliği önler.

Bir asenkron üretici fonksiyonu, `await` ifadelerinin yanı sıra `async for` ve `async with` ifadeleri içerebilir.

asenkron jeneratör yineleyici Bir *asynchronous generator* işlevi tarafından oluşturulan bir nesne.

Bu, `__anext__()` yöntemi kullanılarak çağrıldığında, bir sonraki `yield` ifadesine kadar *asynchronous generator* işlevinin gövdesini yürütecek, beklenebilir bir nesne döndüren bir *asynchronous iterator*.

Her `yield`, konum yürütme durumunu hatırlayarak (yerel değişkenler ve bekleyen `try` ifadeleri dahil) işlemeyi geçici olarak askıya alır. *asynchronous generator iterator*, `__anext__()` tarafından döndürülen başka bir beklenebilir ile etkili bir şekilde devam ettiğinde, kaldığı yerden devam eder. Bkz. **PEP 492** ve **PEP 525**.

eşzamansız yinelenebilir Bir `async for` ifadesinde kullanılabilen bir nesne. `__aiter__()` yönteminden bir *asynchronous iterator* döndürmelidir. **PEP 492** 'de tanımlandı.

asenkron yineleyici 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**.

nitelik A value associated with an object which is referenced by name using dotted expressions. For example, if an object *o* has an attribute *a* it would be referenced as *o.a*.

beklenebilir `await` ifadesinde kullanılabilen bir nesne. Bir *coroutine* veya `__await__()` yöntemine sahip bir nesne olabilir. Ayrıca bakınız **PEP 492**.

BDFL Benevolent Dictator For Life, namı diğer Guido van Rossum, Python'un yaratıcısı.

ikili dosya Bir *dosya nesnesi* *bayt benzeri nesneler* okuyabilir ve yazabilir. İkili dosya örnekleri, ikili modda açılan dosyalardır ('rb', 'wb' veya 'rb+'), `sys.stdin.buffer`, `sys.stdout.buffer` ve `io.BytesIO` ve `gzip.GzipFile` örnekleri.

Ayrıca `str` nesnelerini okuyabilen ve yazabilen bir dosya nesnesi için *text file* 'a bakın.

bayt benzeri nesne *Buffer Protocol* 'i destekleyen ve bir *C-contiguous* arabelleğini dışa aktarabilen bir nesne. Bu, tüm `bytes`, `bytearray` ve `array.array` nesnelerinin yanı sıra birçok yaygın *memoryview* nesnesini içerir. Bayt benzeri nesneler, ikili verilerle çalışan çeşitli işlemler için kullanılabilir; bunlara sıkıştırma, ikili dosyaya kaydetme ve bir soket üzerinden gönderme dahildir.

Bazı işlemler, değişken olması için ikili verilere ihtiyaç duyar. Belgeler genellikle bunlara "okuma-yazma bayt benzeri nesneler" olarak atıfta bulunur. Örnek değiştirilebilir arabellek nesneleri `bytearray` ve bir `bytearray memoryview` içerir. Diğer işlemler, ikili verilerin değişmez nesnelerde ("salt okunur bayt

benzeri nesneler”) depolanmasını gerektirir; bunların örnekleri arasında `bytes` ve bir `bytes` nesnesinin `memoryview` bulunur.

bayt kodu Python kaynak kodu, bir Python programının CPython yorumlayıcısındaki dahili temsili olan bayt kodunda derlenir. Bayt kodu ayrıca `.pyc` dosyalarında önbelleğe alınır, böylece aynı dosyanın ikinci kez çalıştırılması daha hızlı olur (kaynaktan bayt koduna yeniden derleme önlenir). Bu “ara dilin”, her bir bayt koduna karşılık gelen makine kodunu yürüten bir *sanal makine* üzerinde çalıştığı söylenir. Bayt kodlarının farklı Python sanal makineleri arasında çalışması veya Python sürümleri arasında kararlı olması beklenmediğini unutmayın.

Bayt kodu talimatlarının bir listesi `bytecodes` dokümanında bulunabilir.

geri çağırmak Gelecekte bir noktada yürütülecek bir argüman olarak iletilen bir alt program işlevi.

sınıf Kullanıcı tanımlı nesneler oluşturmak için bir şablon. Sınıf tanımları normalde sınıfın örnekleri üzerinde çalışan yöntem tanımlarını içerir.

sınıf değişkeni Bir sınıfta tanımlanmış ve yalnızca sınıf düzeyinde (yani sınıfın bir örneğinde değil) değiştirilmesi amaçlanan bir değişken.

zorlama Aynı türden iki argüman içeren bir işlem sırasında bir tür örneğinin diğerine örtük olarak dönüştürülmesi. Örneğin, `int(3.15)`, kayan noktalı sayıyı 3 tamsayısına dönüştürür, ancak `3+4.5`’te her argüman farklı türdedir (bir `int`, bir kayan nokta), ve her ikisi de eklenmeden önce aynı türe dönüştürülmelidir, aksi takdirde bir `TypeError` yükseltir. Zorlama olmadan, uyumlu türlerin bile tüm argümanlarının programcı tarafından aynı değere normalleştirilmesi gerekir, örneğin: `3+4, 5` yerine `float(3)+4, 5`.

karmaşık sayı Tüm sayıların bir reel kısım ve bir sanal kısım toplamı olarak ifade edildiği bilinen gerçek sayı sisteminin bir uzantısı. Hayali sayılar, hayali birimin gerçek katlarıdır (-1 ’in karekökü), genellikle matematikte i veya mühendislikte j ile yazılır. Python, bu son gösterimle yazılan karmaşık sayılar için yerleşik desteğe sahiptir; hayali kısım bir j son ekiyle yazılır, örneğin `3+1j`. `math` modülünün karmaşık eşdeğerlerine erişmek için `cmath` kullanın. Karmaşık sayıların kullanımı oldukça gelişmiş bir matematiksel özelliktir. Onlara olan ihtiyacın farkında değilseniz, onları güvenli görmezden gelebileceğiniz neredeyse kesindir.

bağlam yöneticisi `with` ifadesinde görülen ortamı `__enter__()` ve `__exit__()` yöntemlerini tanımlayarak kontrol eden bir nesne. Bakınız [PEP 343](#).

bağlam değişkeni Bağlamına bağlı olarak farklı değerler alabilen bir değişken. Bu, her yürütme iş parçasığının bir değişken için farklı bir değere sahip olabileceği Thread-Local Storage’a benzer. Bununla birlikte, bağlam değişkenleriyle, bir yürütme iş parçasığında birkaç bağlam olabilir ve bağlam değişkenlerinin ana kullanımı, eşzamanlı zaman uyumsuz görevlerde değişkenleri izlemektir. Bakınız `contextvars`.

bitişik Bir arabellek, *C-bitişik* veya *Fortran bitişik* ise tam olarak bitişik olarak kabul edilir. Sıfır boyutlu arabellekler C ve Fortran bitişiktir. Tek boyutlu dizilerde, öğeler sıfırdan başlayarak artan dizinler sırasına göre bellekte yan yana yerleştirilmelidir. Çok boyutlu C-bitişik dizilerde, öğeleri bellek adresi sırasına göre ziyaret ederken son dizin en hızlı şekilde değişir. Ancak, Fortran bitişik dizilerinde, ilk dizin en hızlı şekilde değişir.

eşyordam Eşyordamlar, altordamların daha genelleştirilmiş bir biçimidir. Alt programlara bir noktada girilir ve başka bir noktada çıkılır. Eşyordamlar birçok farklı noktada girilebilir, çıkılabilir ve devam ettirilebilir. `async def` ifadesi ile uygulanabilirler. Ayrıca bakınız [PEP 492](#).

eşyordam işlevi Bir *coroutine* nesnesi döndüren bir işlev. Bir eşyordam işlevi `async def` ifadesiyle tanımlanabilir ve `await`, `async for` ve `async with` anahtar kelimelerini içerebilir. Bunlar [PEP 492](#) tarafından tanıtıldı.

CPython Python programlama dilinin [python.org](#) üzerinde dağıtıldığı şekilde kurallı uygulaması. “CPython” terimi, gerektiğinde bu uygulamayı Jython veya IronPython gibi diğerlerinden ayırmak için kullanılır.

dekoratör Genellikle `@wrapper` sözdizimi kullanılarak bir işlev dönüşümü olarak uygulanan, başka bir işlevi döndüren bir işlev. Dekoratörler için yaygın örnekler şunlardır: `classmethod()` ve `staticmethod()`.

Dekoratör sözdizimi yalnızca sözdizimsel şekerdir, aşağıdaki iki işlev tanımları anlamsal olarak eşdeğerdir:

```
def f(arg):
    ...
f = staticmethod(f)
```

(continues on next page)

```
@staticmethod
def f(arg):
    ...
```

Aynı kavram sınıflar için de mevcuttur, ancak orada daha az kullanılır. Dekoratörler hakkında daha fazla bilgi için function definitions ve class definitions belgelerine bakın.

tanımlayıcı `__get__()`, `__set__()` veya `__delete__()` yöntemlerini tanımlayan herhangi bir nesne. Bir sınıf özniteliği bir tanımlayıcı olduğunda, öznitelik araması üzerine özel bağlama davranışı tetiklenir. Normalde, bir özniteliği almak, ayarlamak veya silmek için *a.b* kullanmak, *a* için sınıf sözlüğünde *b* adlı nesneyi arar, ancak *b* bir tanımlayıcı ise, ilgili tanımlayıcı yöntemi çağrılır. Tanımlayıcıları anlamak, Python'u derinlemesine anlamamanın anahtarıdır çünkü bunlar, işlevler, yöntemler, özellikler, sınıf yöntemleri, statik yöntemler ve süper sınıflara başvuru gibi birçok özelliğin temelidir.

Tanımlayıcıların yöntemleri hakkında daha fazla bilgi için, bkz. descriptors veya Descriptor How To Guide.

sözlük Rasgele anahtarların değerlerle eşlendiği ilişkisel bir dizi. Anahtarlar, `__hash__()` ve `__eq__()` yöntemleriyle herhangi bir nesne olabilir. Perl'de karma denir.

sözlük anlama Öğelerin tümünü veya bir kısmını yinelenebilir bir şekilde işlemenin ve sonuçları içeren bir sözlük döndürmenin kompakt bir yolu. `results = {n: n ** 2 for range(10)}`, `n ** 2` değerine eşlenmiş *n* anahtarını içeren bir sözlük oluşturur. Bkz. comprehensions.

sözlük görünümü `dict.keys()`, `dict.values()` ve `dict.items()` 'den döndürülen nesnelere sözlük görünümüleri denir. Sözlüğün girişleri üzerinde dinamik bir görünüm sağlarlar; bu, sözlük değiştiğinde görünümün bu değişiklikleri yansıttığı anlamına gelir. Sözlük görünümünü tam liste olmaya zorlamak için `list(dictview)` kullanın. Bakınız dict-views.

belge dizisi Bir sınıf, işlev veya modülde ilk ifade olarak görünen bir dize değişmez. Paket yürütüldüğünde yoksayılırken, derleyici tarafından tanınır ve çevreleyen sınıfın, işlevin veya modülün `__doc__` özniteliğine yerleştirilir. İç gözetim yoluyla erişilebilir olduğundan, nesnenin belgelenmesi için kurallı yerdir.

duck-typing Doğru arayüze sahip olup olmadığını belirlemek için bir nesnenin türüne bakmayan bir programlama stili; bunun yerine, yöntem veya nitelik basitçe çağrılır veya kullanılır ("Ördek gibi görünüyorsa ve ördek gibi vaklıyorsa, ördek olmalıdır.") İyi tasarlanmış kod, belirli türlerden ziyade arayüzleri vurgulayarak, polimorfik ikameye izin vererek esnekliğini artırır. Ördek yazma, `type()` veya `isinstance()` kullanan testleri önler. (Ancak, ördek yazmanın *abstract base class* ile tamamlanabileceğini unutmayın.) Bunun yerine, genellikle `hasattr()` testleri veya *EAFP* programlamasını kullanır.

EAFP Af dilemek izin almaktan daha kolaydır. Bu yaygın Python kodlama stili, geçerli anahtarların veya niteliklerin varlığını varsayar ve varsayımın yanlış çıkması durumunda istisnaları yakalar. Bu temiz ve hızlı stil, birçok `try` ve `except` ifadesinin varlığı ile karakterize edilir. Teknik, C gibi diğer birçok dilde ortak olan *LBYL* stiliyle çelişir.

ifade (değer döndürür) Bir değere göre değerlendirilebilecek bir sözdizimi parçası. Başka bir deyişle, bir ifade, tümü bir değer döndüren sabit değerler, adlar, öznitelik erişimi, işleçler veya işlev çağrılar gibi ifade öğelerinin bir toplamıdır. Diğer birçok dilin aksine, tüm dil yapıları ifade değildir. Ayrıca `while` gibi kullanılamayan *ifadeler* de vardır. Atamalar da değer döndürmeyen ifadelerdir (statement).

uzatma modülü Çekirdekle ve kullanıcı koduyla etkileşim kurmak için Python'un C API'sini kullanan, C veya C++ ile yazılmış bir modül.

f-string Ön eki 'f' veya 'F' olan dize değişmezleri genellikle "f-strings" olarak adlandırılır; bu, formatted string literals 'ın kısaltmasıdır. Ayrıca bkz. [PEP 498](#).

dosya nesnesi Dosya yönelimli bir API'yi (`read()` veya `write()` gibi yöntemlerle) temel alınan bir kaynağa gösteren bir nesne. Oluşturulma şekline bağlı olarak, bir dosya nesnesi gerçek bir disk üzerindeki dosyaya veya başka bir tür depolama veya iletişim aygıtına (örneğin standart giriş/çıkış, bellek içi arabellekler, yuvalar, borular vb.) erişime aracılık edebilir. Dosya nesneleri ayrıca *file-like objects* veya *streams* olarak da adlandırılır.

Aslında üç dosya nesnesi kategorisi vardır: ham *binary files*, arabelleğe alınmış *binary files* ve *text files*. Arayüzleri `io` modülünde tanımlanmıştır. Bir dosya nesnesi yaratmanın kurallı yolu `open()` işlevini kullanmaktır.

dosya benzeri nesne *dosya nesnesi* ile eşanlamlıdır.

bulucu İçer aktarılmakta olan bir modül için *loader* 'ı bulmaya çalışan bir nesne.

Python 3.3'ten beri, iki çeşit bulucu vardır: `sys.meta_path` ile kullanılmak üzere *meta yol bulucular*, ve `sys.path_hooks` ile kullanılmak üzere *yol girişi bulucular*.

Daha fazla ayrıntı için **PEP 302**, **PEP 420** ve **PEP 451** bakın.

kat bölümü En yakın tam sayıya yuvarlayan matematiksel bölme. Kat bölme operatörü `//` şeklindedir. Örneğin, `11 // 4` ifadesi, gerçek yüzer bölme tarafından döndürülen `2.75` değerinin aksine `2` olarak değerlendirilir. `(-11) // 4` 'ün `-3` olduğuna dikkat edin, çünkü bu `-2.75` yuvarlatılmış *aşağı*. Bakınız **PEP 238**.

fonksiyon Bir araya bir değer döndüren bir dizi ifade. Ayrıca, gövdenin yürütülmesinde kullanılabilen sıfır veya daha fazla *argüman* iletebilir. Ayrıca *parameter*, *method* ve *function* bölümüne bakın.

fonksiyon açıklaması Bir işlev parametresinin veya dönüş değerinin *ek açıklaması*.

İşlev ek açıklamaları genellikle *type hints* için kullanılır: örneğin, bu fonksiyonun iki `int` argüman alması ve ayrıca bir `int` dönüş değerine sahip olması beklenir

```
def sum_two_numbers(a: int, b: int) -> int:
    return a + b
```

İşlev açıklama sözdizimi *function* bölümünde açıklanmaktadır.

See *variable annotation* and **PEP 484**, which describe this functionality.

__future__ Bir *future* ifadesi, `from __future__ import <feature>`, derleyiciyi, Python'un gelecekteki bir sürümünde standart hale gelecek olan sözdizimini veya semantiği kullanarak mevcut modülü derlemeye yönlendirir. `__future__` modülü, *feature*'ın olası değerlerini belgeler. Bu modülü içe aktararak ve değişkenlerini değerlendirerek, dile ilk kez yeni bir özelliğin ne zaman eklendiğini ve ne zaman varsayılan olacağını (ya da yaptığını) görebilirsiniz:

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

çöp toplama Artık kullanılmadığında belleği boşaltma işlemi. Python, referans sayımı ve referans döngülerini algılayıp kırabilen bir döngüsel çöp toplayıcı aracılığıyla çöp toplama gerçekleştirir. Çöp toplayıcı `gc` modülü kullanılarak kontrol edilebilir.

jeneratör Bir *generator iterator* döndüren bir işlev. Bir *for* döngüsünde kullanılabilen bir dizi değer üretmek için `yield` ifadeleri içermesi veya `next()` işleviyle birer birer alınabilmesi dışında normal bir işleve benziyor.

Genellikle bir üretici işlevine atıfta bulunur, ancak bazı bağlamlarda bir *jeneratör yineleyicisine* atıfta bulunabilir. Amaçlanan anlamın net olmadığı durumlarda, tam terimlerin kullanılması belirsizliği önler.

jeneratör yineleyici Bir *generator* işlevi tarafından oluşturulan bir nesne.

Her `yield`, konum yürütme durumunu hatırlayarak (yerel değişkenler ve bekleyen `try` ifadeleri dahil) işlemeyi geçici olarak askıya alır. *jeneratör yineleyici* devam ettiğinde, kaldığı yerden devam eder (her çağrıda yeniden başlayan işlevlerin aksine).

jeneratör ifadesi Yineleyici döndüren bir ifade. Bir döngü değişkenini, aralığı ve isteğe bağlı bir `if` yan tümcesini tanımlayan bir `for` yan tümcesinin takip ettiği normal bir ifadeye benziyor. Birleştirilmiş ifade, bir çevreleyen için değerler üretir:

```
>>> sum(i*i for i in range(10))           # sum of squares 0, 1, 4, ... 81
285
```

genel işlev Farklı türler için aynı işlemi uygulayan birden çok işlevden oluşan bir işlev. Bir çağrı sırasında hangi uygulamanın kullanılması gerektiği, gönderme algoritması tarafından belirlenir.

Ayrıca *single dispatch* sözlük girdisine, `functools singledispatch()` dekoratörüne ve **PEP 443** 'e bakın.

genel tip Parametrelendirilebilen bir *type*; tipik olarak bir konteyner sınıfı, örneğin `list` veya `dict`. *type hint* ve *annotation* için kullanılır.

Daha fazla ayrıntı için generic alias types, [PEP 483](#), [PEP 484](#), [PEP 585](#) ve `typing` modülüne bakın.

GIL Bakınız *global interpreter lock*.

genel tercüman kilidi *CPython* yorumlayıcısı tarafından aynı anda yalnızca bir iş parçasığının Python *bytecode* 'u yürütmesini sağlamak için kullanılan mekanizma. Bu, nesne modelini (`dict` gibi kritik yerleşik türler dahil) eşzamanlı erişime karşı örtük olarak güvenli hale getirerek *CPython* uygulamasını basitleştirir. Tüm yorumlayıcıyı kilitlemek, çok işlemcili makinelerin sağladığı paralelliğin çoğu pahasına, yorumlayıcının çok iş parçasıklı olmasını kolaylaştırır.

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.

“Serbest iş parçasıklı” bir yorumlayıcı (paylaşılan verileri çok daha ince bir ayrıntı düzeyinde kilitleyen) oluşturma çabaları, ortak tek işlemcili durumda performans düştüğü için başarılı olmamıştır. Bu performans sorununun üstesinden gelinmesinin uygulamayı çok daha karmaşık hale getireceğine ve dolayısıyla bakımını daha maliyetli hale getireceğine inanılmaktadır.

karma tabanlı pyc Geçerliliğini belirlemek için ilgili kaynak dosyanın son değiştirilme zamanı yerine karma değerini kullanan bir bayt kodu önbellek dosyası. Bakınız `pyc-invalidation`.

yıkanabilir Bir nesne, ömrü boyunca asla değişmeyen bir karma değere sahipse (bir `__hash__()` yöntemine ihtiyaç duyar) ve diğer nesnelerle karşılaştırılabilirse (bir `__eq__()` yöntemine ihtiyaç duyar) *hashable* olur. Eşit karşılaştıran *Hashable* nesneleri aynı karma değerine sahip olmalıdır.

Hashability, bir nesneyi bir sözlük anahtarı ve bir set üyesi olarak kullanılabilir hale getirir, çünkü bu veri yapıları hash değerini dahili olarak kullanır.

Python'un değişmez yerleşik nesnelerinin çoğu, yıkanabilir; değiştirilebilir kaplar (listeler veya sözlükler gibi) değildir; değişmez kaplar (tüpler ve donmuş kümeler gibi) yalnızca öğelerinin yıkanabilir olması durumunda yıkanabilir. Kullanıcı tanımlı sınıfların örnekleri olan nesneler varsayılan olarak hash edilebilir. Hepsini eşit olmayı karşılaştırır (kendileriyle hariç) ve hash değerleri `id()` 'lerinden türetilir.

BOŞTA An Integrated Development Environment for Python. IDLE is a basic editor and interpreter environment which ships with the standard distribution of Python.

değişmez Sabit değeri olan bir nesne. Değişmez nesneler arasında sayılar, dizeler ve demetler bulunur. Böyle bir nesne değiştirilemez. Farklı bir değer saklanması gerekiyorsa yeni bir nesne oluşturulmalıdır. Örneğin bir sözlükte anahtar olarak, sabit bir karma değerinin gerekli olduğu yerlerde önemli bir rol oynarlar.

içe aktarım yolu İçe aktarılacak modüller için *path based finder* tarafından aranan konumların (veya *path entries*) listesi. İçe aktarma sırasında, bu konum listesi genellikle `sys.path` adresinden gelir, ancak alt paketler için üst paketin `__path__` özelliğinden de gelebilir.

içe aktarma Bir modüldeki Python kodunun başka bir modüldeki Python koduna sunulması süreci.

içe aktarıcı Bir modülü hem bulan hem de yükleyen bir nesne; hem bir *finder* hem de *loader* nesnesi.

etkileşimli Python'un etkileşimli bir yorumlayıcısı vardır; bu, yorumlayıcı isteminde ifadeler ve ifadeler girebileceğiniz, bunları hemen çalıştırabileceğiniz ve sonuçlarını görebileceğiniz anlamına gelir. Herhangi bir argüman olmadan `python` 'u başlatmanız yeterlidir (muhtemelen bilgisayarınızın ana menüsünden seçerek). Yeni fikirleri test etmenin veya modülleri ve paketleri incelemenin çok güçlü bir yoludur (`help(x)` 'i unutmayın).

yorumlanmış Python, derlenmiş bir dilin aksine yorumlanmış bir dildir, ancak bayt kodu derleyicisinin varlığı nedeniyle ayırım bulanık olabilir. Bu, kaynak dosyaların daha sonra çalıştırılacak bir yürütülebilir dosya oluşturmada doğrudan çalıştırılabilirliği anlamına gelir. Yorumlanan diller genellikle derlenmiş dillerden daha kısa bir geliştirme/hata ayıklama döngüsüne sahiptir, ancak programları genellikle daha yavaş çalışır. Ayrıca bkz. *interactive*.

tercüman kapatma Kapatılması istendiğinde, Python yorumlayıcısı, modüller ve çeşitli kritik iç yapılar gibi tahsis edilen tüm kaynakları kademeli olarak serbest bıraktığı özel bir aşamaya girer. Ayrıca *garbage collector* için

birkaç çağrı yapar. Bu, kullanıcı tanımlı yıkıcılarda veya zayıf referans geri aramalarında kodun yürütülmesini tetikleyebilir. Kapatma aşamasında yürütülen kod, dayandığı kaynaklar artık çalışmayabileceğinden çeşitli istisnalarla karşılaşabilir (yaygın örnekler kitaplık modülleri veya uyarı makineleridir).

Yorumlayıcının kapatılmasının ana nedeni, `__main__` modülünün veya çalıştırılan betiğin yürütmeyi bitirmiş olmasıdır.

yinelenebilir Üyelerini teker teker döndürebilen bir nesne. Yineleme örnekleri, tüm dizi türlerini (`list`, `str`, and `tuple` gibi) ve `dict`, *dosya objeleri* gibi bazı dizi olmayan türleri ve bir `__iter__()` yöntemiyle veya *dizi* semantiğini uygulayan bir `__getitem__()` yöntemiyle tanımladığınız tüm sınıfların nesnelerini içerir.

Yinelenebilirler bir `for` döngüsünde ve bir dizinin gerekli olduğu diğer birçok yerde kullanılabilir (`zip()`, `map()`, ...). Yerleşik `iter()` işlevine argüman olarak yinelenebilir bir nesne iletildiğinde, nesne için bir yineleyici döndürür. Bu yineleyici, değerler kümesi üzerinden bir geçiş için iyidir. Yinelenebilirleri kullanırken, genellikle `iter()` çağırmanız veya yineleyici nesnelerle kendiniz ilgilenmeniz gerekmez. `for` ifadesi bunu sizin için otomatik olarak yapar ve yineleyiciyi döngü süresince tutmak için geçici bir adsız değişken oluşturur. Ayrıca bkz. *iterator*, *sequence* ve *generator*.

yineleyici Bir veri akışını temsil eden bir nesne. Yineleyicinin `__next__()` yöntemine (veya yerleşik `next()` işlevine iletilmesi) yinelenen çağrılar, akıştaki ardışık öğeleri döndürür. Daha fazla veri bulunmadığında, bunun yerine bir `StopIteration` istisnası oluşturulur. Bu noktada, yineleyici nesnesi tükenir ve `__next__()` yöntemine yapılan diğer çağrılar yalnızca `StopIteration` ögesini yeniden yükseltir. Yineleyicilerin, yineleyici nesnesinin kendisini döndüren bir `__iter__()` yöntemine sahip olmaları gerekir, böylece her yineleyici de yinelenebilir ve diğer yinelenebilirlerin kabul edildiği çoğu yerde kullanılabilir. Dikkate değer bir istisna, birden çok yineleme geçişini deneyen koddur. Bir kapsayıcı nesnesi (örneğin bir `list`), onu `iter()` işlevine her ilettiğinizde veya onu bir `for` döngüsünde kullandığınızda yeni bir yineleyici üretir. Bunu bir yineleyiciyle denemek, önceki yineleme geçişinde kullanılan aynı tükenmiş yineleyici nesnesini döndürerek boş bir kap gibi görünmesini sağlar.

Daha fazla bilgi `typeiter` içinde bulunabilir.

anahtar işlev Anahtar işlevi veya harmanlama işlevi, sıralama veya sıralama için kullanılan bir değeri döndüren bir çağrılabilir. Örneğin, `locale.strxfrm()`, yerel ayara özgü sıralama kurallarının farkında olan bir sıralama anahtarı üretmek için kullanılır.

Python'daki bir dizi araç, öğelerin nasıl sıralandığını veya gruplandırıldığını kontrol etmek için temel işlevleri kabul eder. Bunlar `min()`, `max()`, `sorted()`, `list.sort()`, `heapq.merge()`, `heapq.nsmallest()`, `heapq.nlargest()` ve `itertools.groupby()`.

Bir tuş işlevi oluşturmanın birkaç yolu vardır. Örneğin, `str.lower()` yöntemi, büyük/küçük harfe duyarlı olmayan sıralamalar için bir anahtar işlevi işlevi görebilir. Alternatif olarak, `lambda r: (r[0], r[2])` gibi bir `lambda` ifadesinden bir anahtar işlevi oluşturulabilir. Ayrıca, `operator` modülü üç temel işlev kurucusu sağlar: `attrgetter()`, `itemgetter()` ve `methodcaller()`. Anahtar işlevlerin nasıl oluşturulacağı ve kullanılacağına ilişkin örnekler için *Sorting HOW TO* bölümüne bakın.

anahtar kelime argümanı Bakınız *argument*.

lambda İşlev çağrıldığında değerlendirilen tek bir *expression* 'dan oluşan anonim bir satır içi işlev. Bir `lambda` işlevi oluşturmak için sözdizimi `lambda [parametreler]: ifade` şeklindedir

LBYL Zıplamadan önce Bak. Bu kodlama stili, arama veya arama yapmadan önce ön koşulları açıkça test eder. Bu stil, *EAFP* yaklaşımıyla çelişir ve birçok `if` ifadesinin varlığı ile karakterize edilir.

Çok iş parçacıklı bir ortamda, LBYL yaklaşımı “bakan” ve “sıçrayan” arasında bir yarış koşulu getirme riskini taşıyabilir. Örneğin, `if key in mapping: return mapping[key]` kodu, testten sonra, ancak aramadan önce başka bir iş parçacığı *eşlemeden* `key` kaldırırsa başarısız olabilir. Bu sorun, kilitlerle veya *EAFP* yaklaşımı kullanılarak çözülebilir.

liste Yerleşik bir Python *dizi*. Adına rağmen, öğelere erişim $O(1)$ olduğundan, diğer dillerdeki bir diziye, bağlantılı bir listeden daha yakındır.

liste anlama Bir dizideki öğelerin tümünü veya bir kısmını işlemenin ve sonuçları içeren bir liste döndürmenin kompakt bir yolu. `sonuç = ['{:04x}'.format(x) for range(256) if x % 2 == 0]`, dizinde çift onaltılık sayılar (0x..) içeren bir diziler listesi oluşturur. 0 ile 255 arasındadır. `if` yan tümcesi isteğe bağlıdır. Atlanırsa, “aralık(256)” içindeki tüm öğeler işlenir.

yükleyici Modül yükleyen bir nesne. `load_module()` adında bir yöntem tanımlanmalıdır. Bir yükleyici genellikle bir *finder* ile döndürülür. Ayrıntılar için [PEP 302](#) ve bir *soyut temel sınıf* için `importlib.abc.Loader` bölümüne bakın.

sihirli yöntem *special method* için gayri resmi bir eşanlamı.

haritalama Keyfi anahtar aramalarını destekleyen ve Mapping veya MutableMapping collections-abstract-base-classes içinde belirtilen yöntemleri uygulayan bir kapsayıcı nesnesi temel sınıflar. Örnekler arasında `dict`, `collections.defaultdict`, `collections.OrderedDict` ve `collections.Counter` sayılabilir.

meta yol bulucu Bir *finder*, `sys.meta_path` aramasıyla döndürülür. Meta yol bulucular, *yol girişi bulucuları* ile ilişkilidir, ancak onlardan farklıdır.

Meta yol bulucuların uyguladığı yöntemler için `importlib.abc.MetaPathFinder` bölümüne bakın.

metasınıf Bir sınıfın sınıfı. Sınıf tanımları, bir sınıf adı, bir sınıf sözlüğü ve temel sınıfların bir listesini oluşturur. Metasınıf, bu üç argümanı almaktan ve sınıfı oluşturmaktan sorumludur. Çoğu nesne yönelimli programlama dili, varsayılan bir uygulama sağlar. Python'u özel yapan şey, özel metasınıflar oluşturma mümkün olmasıdır. Çoğu kullanıcı bu araca hiçbir zaman ihtiyaç duymaz, ancak ihtiyaç duyulduğunda, metasınıflar güçlü ve zarif çözümler sağlayabilir. Nitelik erişimini günlüğe kaydetmek, iş parçacığı güvenliği eklemek, nesne oluşturmaya izlemek, tekilleri uygulamak ve diğer birçok görev için kullanılmışlardır.

Daha fazla bilgi metaclasses içinde bulunabilir.

metot Bir sınıf gövdesi içinde tanımlanan bir işlev. Bu sınıfın bir örneğinin özniteliği olarak çağrılırsa, yöntem örnek nesnesini ilk *argument* (genellikle `self` olarak adlandırılır) olarak alır. Bkz. *function* ve *nested scope*.

metot kalite sıralaması Metot Çözüm Sırası, arama sırasında bir üye için temel sınıfların arandığı sıradır. 2.3 sürümünden bu yana Python yorumlayıcısı tarafından kullanılan algoritmanın ayrıntıları için bkz. [The Python 2.3 Method Resolution Order](#)

modül Python kodunun kuruluş birimi olarak hizmet eden bir nesne. Modüller, rastgele Python nesneleri içeren bir ad alanına sahiptir. Modüller, *importing* işlemiyle Python'a yüklenir.

Ayrıca bakınız *package*.

modül özelliği Bir modülü yüklemek için kullanılan içe aktarmayla ilgili bilgileri içeren bir ad alanı. Bir `importlib.machinery.ModuleSpec` örneği.

MRO Bakınız *metot çözüm sırası*.

değiştirilebilir Değiştirilebilir (mutable) nesneler değerlerini değiştirebilir ancak idlerini koruyabilirler. Ayrıca bkz. *immutable*.

adlandırılmış demet “named tuple” terimi, demetten miras alan ve dizinlenebilir öğelerine de adlandırılmış nitelikler kullanılarak erişilebilen herhangi bir tür veya sınıf için geçerlidir. Tür veya sınıfın başka özellikleri de olabilir.

Çeşitli yerleşik türler, `time.localtime()` ve `os.stat()` tarafından döndürülen değerler de dahil olmak üzere, tanımlama grupları olarak adlandırılır. Başka bir örnek `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
```

Bazı adlandırılmış demetler yerleşik türlerdir (yukarıdaki örnekler gibi). Alternatif olarak, tuple öğesinden miras alan ve adlandırılmış alanları tanımlayan normal bir sınıf tanımından adlandırılmış bir tanımlama grubu oluşturulabilir. Böyle bir sınıf elle yazılabilir veya fabrika işlevi `collections.namedtuple()` ile oluşturulabilir. İkinci teknik ayrıca elle yazılmış veya yerleşik adlandırılmış demetlerde bulunmayan bazı ekstra yöntemler ekler.

ad alanı Değişkenin saklandığı yer. Ad alanları sözlükler olarak uygulanır. Nesnelerde (yöntemlerde) yerel, genel ve yerleşik ad alanlarının yanı sıra iç içe ad alanları vardır. Ad alanları, adlandırma çakışmalarını önleyerek

modülerliği destekler. Örneğin, `builtins.open` ve `os.open()` işlevleri ad alanlarıyla ayırt edilir. Ad alanları, hangi modülün bir işlevi uyguladığını açıkça belirterek okunabilirliğe ve sürdürülebilirliğe de yardımcı olur. Örneğin, `random.seed()` veya `itertools.islice()` yazmak, bu işlevlerin sırasıyla `random` ve `itertools` modülleri tarafından uygulandığını açıkça gösterir.

ad alanı paketi A **PEP 420 package**, yalnızca alt paketler için bir kap olarak hizmet eder. Ad alanı paketlerinin hiçbir fiziksel temsili olmayabilir ve `__init__.py` dosyası olmadığından özellikle *regular package* gibi değildirler.

Ayrıca bkz. *module*.

iç içe kapsam Kapsamlı bir tanımdaki bir değişkene atıfta bulunma yeteneği. Örneğin, başka bir fonksiyonun içinde tanımlanan bir fonksiyon, dış fonksiyondaki değişkenlere atıfta bulunabilir. İç içe kapsamların varsayılan olarak yalnızca başvuru için çalıştığını ve atama için çalışmadığını unutmayın. Yerel değişkenler en içteki kapsamda hem okur hem de yazar. Benzer şekilde, global değişkenler global ad alanını okur ve yazar. `nonlocal`, dış kapsamlara yazmaya izin verir.

yeni stil sınıf Artık tüm sınıf nesneleri için kullanılan sınıfların lezzetinin eski adı. Önceki Python sürümlerinde, yalnızca yeni stil sınıfları Python'un `__slots__`, tanımlayıcılar, özellikler, `__getattr__()`, sınıf yöntemleri ve statik yöntemler gibi daha yeni, çok yönlü özelliklerini kullanabilirdi.

obje Durum (öznitelikler veya değer) ve tanımlanmış davranış (yöntemler) içeren herhangi bir veri. Ayrıca herhangi bir *yeni tarz sınıfın* nihai temel sınıfı.

paket A Python *module* which can contain submodules or recursively, subpackages. Technically, a package is a Python module with an `__path__` attribute.

Ayrıca bkz. *regular package* ve *namespace package*.

parametre Bir *function* (veya yöntem) tanımında, işlevin kabul edebileceği bir *argument* (veya bazı durumlarda, argümanlar) belirten adlandırılmış bir varlık. Beş çeşit parametre vardır:

- *positional-or-keyword*: *pozisyonel* veya bir *keyword argümanı* olarak iletilebilen bir argüman belirtir. Bu, varsayılan parametre türüdür, örneğin aşağıdakilerde *foo* ve *bar*:

```
def func(foo, bar=None): ...
```

- *positional-only*: yalnızca konuma göre sağlanabilen bir argüman belirtir. Yalnızca konumsal parametreler, onlardan sonra fonksiyon tanımının parametre listesine bir / karakteri eklenerek tanımlanabilir, örneğin aşağıdakilerde *posonly1* ve *posonly2*:

```
def func(posonly1, posonly2, /, positional_or_keyword): ...
```

- *keyword-only*: sadece anahtar kelime ile sağlanabilen bir argüman belirtir. Yalnızca anahtar kelime (keyword-only) parametreleri, onlardan önceki fonksiyon tanımının parametre listesine tek bir değişken konumlu parametre veya çıplak * dahil edilerek tanımlanabilir, örneğin aşağıdakilerde *kw_only1* ve *kw_only2*:

```
def func(arg, *, kw_only1, kw_only2): ...
```

- *var-positional*: keyfi bir pozisyonel argüman dizisinin sağlanabileceğini belirtir (diğer parametreler tarafından zaten kabul edilmiş herhangi bir konumsal argümana ek olarak). Böyle bir parametre, parametre adının başına * eklenerek tanımlanabilir, örneğin aşağıdakilerde *args*:

```
def func(*args, **kwargs): ...
```

- *var-keyword*: keyfi olarak birçok anahtar kelime argümanının sağlanabileceğini belirtir (diğer parametreler tarafından zaten kabul edilen herhangi bir anahtar kelime argümanına ek olarak). Böyle bir parametre, parametre adının başına **, örneğin yukarıdaki örnekte *kwargs* eklenerek tanımlanabilir.

Parametreler, hem isteğe bağlı hem de gerekli argümanları ve ayrıca bazı isteğe bağlı bağımsız değişkenler için varsayılan değerleri belirtebilir.

Ayrıca bkz. *argüman*, argümanlar ve parametreler arasındaki fark, `inspect.Parameter`, `function` ve **PEP 362**.

yol girişi *path based finder* içe aktarma modüllerini bulmak için başvurduğu *import path* üzerindeki tek bir konum.

yol girişi bulucu Bir *finder* `sys.path_hooks` (yani bir *yol girişi kancası*) üzerinde bir çağrılabilir tarafından döndürülür ve *path entry* verilen modüllerin nasıl bulunacağını bilir.

Yol girişi bulucularının uyguladığı yöntemler için `importlib.abc.PathEntryFinder` bölümüne bakın.

yol girişi kancası `sys.path_hook` listesinde, belirli bir *yol girişindeki* modülleri nasıl bulacağını biliyorsa, bir *yol girişi bulucu* döndüren bir çağrılabilir.

yol tabanlı bulucu Modüller için bir *import path* arayan varsayılan *meta yol buluculardan* biri.

yol benzeri nesne Bir dosya sistemi yolunu temsil eden bir nesne. Yol benzeri bir nesne, bir yolu temsil eden bir `str` veya `bytes` nesnesi veya `os.PathLike` protokolünü uygulayan bir nesnedir. `os.PathLike` protokolünü destekleyen bir nesne, `os.fspath()` işlevi çağrılarak bir `str` veya `bytes` dosya sistemi yoluna dönüştürülebilir; `os.fsdecode()` ve `os.fsencode()`, bunun yerine sırasıyla `str` veya `bytes` sonucunu garanti etmek için kullanılabilir. **PEP 519** tarafından tanıtıldı.

PEP Python Geliştirme Önerisi. PEP, Python topluluğuna bilgi sağlayan veya Python veya süreçleri ya da ortamı için yeni bir özelliği açıklayan bir tasarım belgesidir. PEP'ler, önerilen özellikler için özlü bir teknik şartname ve bir gerekçe sağlamalıdır.

PEP'lerin, önemli yeni özellikler önermek, bir sorun hakkında topluluk girdisi toplamak ve Python'a giren tasarım kararlarını belgelemek için birincil mekanizmalar olması amaçlanmıştır. PEP yazarı, topluluk içinde fikir birliği oluşturmaktan ve muhalif görüşleri belgelemekten sorumludur.

Bakınız **PEP 1**.

kısım **PEP 420** içinde tanımlandığı gibi, bir ad alanı paketine katkıda bulunan tek bir dizindeki (muhtemelen bir zip dosyasında depolanan) bir dizi dosya.

konumsal argüman Bakınız *argument*.

geçici API Geçici bir API, standart kitaplığın geriye dönük uyumluluk garantilerinden kasıtlı olarak hariç tutulan bir API'dir. Bu tür arayüzlerde büyük değişiklikler beklenmese de, geçici olarak işaretlendikleri süreç, çekirdek geliştiriciler tarafından gerekli görüldüğü takdirde geriye dönük uyumsuz değişiklikler (arayüzün kaldırılmasına kadar ve buna kadar) meydana gelebilir. Bu tür değişiklikler karşılıksız yapılmayacaktır - bunlar yalnızca API'nin eklenmesinden önce gözden kaçan ciddi temel kusurlar ortaya çıkarsa gerçekleşecektir.

Geçici API'ler için bile, geriye dönük uyumsuz değişiklikler "son çare çözümü" olarak görülür - tanımlanan herhangi bir soruna geriye dönük uyumlu bir çözüm bulmak için her türlü girişimde bulunulacaktır.

Bu süreç, standart kitaplığın, uzun süreler boyunca sorunlu tasarım hatalarına kilitlenmeden zaman içinde gelişmeye devam etmesini sağlar. Daha fazla ayrıntı için bkz. **PEP 411**.

geçici paket Bakınız *provisional API*.

Python 3000 Python 3.x sürüm satırının takma adı (uzun zaman önce sürüm 3'ün piyasaya sürülmesi uzak bir gelecekte olduğu zaman ortaya çıktı.) Bu aynı zamanda "Py3k" olarak da kısaltılır.

Pythonic Diğer dillerde ortak kavramları kullanarak kod uygulamak yerine Python dilinin en yaygın deyimlerini yakından takip eden bir fikir veya kod parçası. Örneğin, Python'da yaygın bir deyim, bir `for` ifadesi kullanarak yinelenen bir öğenin tüm öğeleri üzerinde döngü oluşturmaktır. Diğer birçok dilde bu tür bir yapı yoktur, bu nedenle Python'a aşina olmayan kişiler bazen bunun yerine sayısal bir sayaç kullanır:

```
for i in range(len(food)) :
    print(food[i])
```

Temizleyicinin aksine, Pythonic yöntemi:

```
for piece in food:
    print(piece)
```

nitelikli isim **PEP 3155** içinde tanımlandığı gibi, bir modülün genel kapsamından o modülde tanımlanan bir sınıfa, işleve veya yönteme giden "yolu" gösteren noktalı ad. Üst düzey işlevler ve sınıflar için nitelikli ad, nesnenin adıyla aynıdır:

```
>>> class C:
...     class D:
...         def meth(self):
...             pass
...
>>> C.__qualname__
'C'
>>> C.D.__qualname__
'C.D'
>>> C.D.meth.__qualname__
'C.D.meth'
```

Modüllere atıfta bulunmak için kullanıldığında, *tam nitelenmiş ad*, herhangi bir üst paket de dahil olmak üzere, module giden tüm noktalı yol anlamına gelir, örn. `email.mime.text`:

```
>>> import email.mime.text
>>> email.mime.text.__name__
'email.mime.text'
```

referans sayısı Bir nesneye yapılan başvuruların sayısı. Bir nesnenin referans sayısı sıfıra düştüğünde, yerinden çıkarılır. Referans sayımı genellikle Python kodunda görülmez, ancak *CPython* uygulamasının önemli bir özge-sidir. `sys` modülü, programcıların belirli bir nesne için referans sayısını döndürmek üzere çağırabilecekleri bir `getrefcount()` işlevini tanımlar.

sürekli paketleme `__init__.py` dosyası içeren bir dizin gibi geleneksel bir *package*.

Ayrıca bkz. *ad alanı paketi*.

__slots__ Örnek öznitelikleri için önceden yer bildirerek ve örnek sözlüklerini ortadan kaldırarak bellekten tasarruf sağlayan bir sınıf içindeki bildirim. Popüler olmasına rağmen, tekniğin doğru olması biraz zor ve en iyi, bellek açısından kritik bir uygulamada çok sayıda örneğin bulunduğu nadir durumlar için ayrılmıştır.

dizi `__getitem__()` özel yöntemi aracılığıyla tamsayı dizinlerini kullanarak verimli öge erişimini destekleyen ve dizinin uzunluğunu döndüren bir `__len__()` yöntemini tanımlayan bir *iterable*. Bazı yerleşik dizi türleri şunlardır: `list`, `str`, `tuple` ve `bytes`. `dict` ayrıca `__getitem__()` ve `__len__()` 'i de desteklediğine dikkat edin, ancak aramalar tamsayılar yerine rastgele *immutable* anahtarları kullandığından bir diziden ziyade bir eşleme olarak kabul edilir.

`collections.abc.Sequence` soyut temel sınıfı; `count()`, `index()`, `__contains__()`, ve `__reversed__()` ekleyerek sadece `__getitem__()` ve `__len__()` 'in ötesine geçen çok daha zengin bir arayüzü tanımlar. Bu genişletilmiş arabirimi uygulayan türler, `register()` kullanılarak açıkça kaydedilebilir.

anlamak Öğelerin tümünü veya bir kısmını yinelenabilir bir şekilde işlemenin ve sonuçlarla birlikte bir küme döndürmenin kompakt bir yolu. `results = {c for c in 'abracadabra' if c not in 'abc'}, {'r', 'd'}` dizelerini oluşturur. Bakınız *comprehensions*.

tek sevk Uygulamanın tek bir argüman türüne göre seçildiği bir *generic function* gönderimi biçimi.

parçalamak Genellikle bir *sequence* 'nin bir bölümünü içeren bir nesne. Bir dilim, örneğin `variable_name[1:3:5]` 'de olduğu gibi, birkaç tane verildiğinde, sayılar arasında iki nokta üst üste koyarak, `[]` alt simge gösterimi kullanılarak oluşturulur. Köşeli ayraç (alt simge) gösterimi, dahili olarak `slice` nesnelerini kullanır.

özel metod Toplama gibi bir tür üzerinde belirli bir işlemi yürütmek için Python tarafından örtük olarak çağrılan bir yöntem. Bu tür yöntemlerin çift alt çizgi ile başlayan ve biten adları vardır. Özel yöntemler *specialnames* içinde belgelenmiştir.

ifade (değer döndürmez) Bir ifade, bir paketin parçasıdır (kod “bloğu”). Bir ifade, bir *expression* veya `if`, `while` veya `for` gibi bir anahtar kelimeye sahip birkaç yapıdan biridir.

yazı çözümleme Python'da bir dize, bir Unicode kod noktaları dizisidir (U+0000–U+10FFFF aralığında). Bir di-ze-yi depolamak veya aktarmak için, bir bayt dizisi olarak seri hale getirilmesi gerekir.

Bir dizeyi bir bayt dizisi halinde seri hale getirmek “kodlama (encoding)” olarak bilinir ve dizeyi bayt dizisinden yeniden oluşturmak “kod çözme (decoding)” olarak bilinir.

Toplu olarak “metin kodlamaları” olarak adlandırılan çeşitli farklı metin serileştirme kodekleri vardır.

yazı dosyası A *file object* str nesnelerini okuyabilir ve yazabilir. Çoğu zaman, bir metin dosyası aslında bir bayt yönelimli veri akışına erişir ve otomatik olarak *text encoding* işler. Metin dosyalarına örnek olarak metin modunda açılan dosyalar ('r' veya 'w'), sys.stdin, sys.stdout ve io.StringIO örnekleri verilebilir.

Ayrıca *ikili dosyaları* okuyabilen ve yazabilen bir dosya nesnesi için *bayt benzeri nesnelere* bakın.

üç tırnaklı dize Üç tırnak işareti (") veya kesme işareti (') ile sınırlanan bir dize. Tek tırnaklı dizelerde bulunmayan herhangi bir işlevsellik sağlamasalar da, birkaç nedenden dolayı faydalıdır. bir dizeye çıkışsız tek ve çift tırnak eklemeniz gerekir ve bunlar, devam karakterini kullanmadan birden çok satıra yayılabilir, bu da onları özellikle belge dizileri yazarken kullanışlı hale getirir.

tip Bir Python nesnesinin türü, onun ne tür bir nesne olduğunu belirler; her nesnenin bir türü vardır. Bir nesnenin tipine `__class__` niteliği ile erişilebilir veya `type(obj)` ile alınabilir.

tip takma adı Bir tanımlayıcıya tür atanarak oluşturulan, bir tür için eş anlamlı.

Tür takma adları, *tür ipuçlarını* basitleştirmek için kullanışlıdır. Örneğin:

```
def remove_gray_shades(
    colors: list[tuple[int, int, int]]) -> list[tuple[int, int, int]]:
    pass
```

bu şekilde daha okunaklı hale getirilebilir:

```
Color = tuple[int, int, int]

def remove_gray_shades(colors: list[Color]) -> list[Color]:
    pass
```

Bu işlevi açıklayan `typing` ve **PEP 484** bölümlerine bakın.

tür ipucu Bir değişken, bir sınıf niteliği veya bir işlev parametresi veya dönüş değeri için beklenen türü belirten bir *ek açıklama*.

Tür ipuçları isteğe bağlıdır ve Python tarafından uygulanmaz, ancak bunlar statik tip analiz araçları için faydalıdır ve kod tamamlama ve yeniden düzenleme ile IDE'lere yardımcı olur.

Genel değişkenlerin, sınıf özniteliklerinin ve işlevlerin tür ipuçlarına, yerel değişkenlere değil, `typing.get_type_hints()` kullanılarak erişilebilir.

Bu işlevi açıklayan `typing` ve **PEP 484** bölümlerine bakın.

evrensel yeni satırlar Aşağıdakilerin tümünün bir satırın bitişi olarak kabul edildiği metin akışlarını yorumlamanın bir yolu: Unix satır sonu kuralı `\n`, Windows kuralı `\r\n`, ve eski Macintosh kuralı `\r`. Ek bir kullanım için **PEP 278** ve **PEP 3116** ve ayrıca `bytes.splitlines()` bakın.

değişken açıklama Bir değişkenin veya bir sınıf özniteliğinin *ek açıklaması*.

Bir değişkene veya sınıf niteliğine açıklama eklerken atama isteğe bağlıdır:

```
class C:
    field: 'annotation'
```

Değişken açıklamaları genellikle *tür ipuçları* için kullanılır: örneğin, bu değişkenin `int` değerlerini alması beklenir:

```
count: int = 0
```

Değişken açıklama sözdizimi `annassign` bölümünde açıklanmıştır.

See *function annotation*, **PEP 484** and **PEP 526**, which describe this functionality.

sanal ortam Python kullanıcılarının ve uygulamalarının, aynı sistem üzerinde çalışan diğer Python uygulamalarının davranışına müdahale etmeden Python dağıtım paketlerini kurmasına ve yükseltmesine olanak tanıyan, işbirliği içinde yalıtılmış bir çalışma zamanı ortamı.

Ayrıca bakınız `venv`.

sanal makine Tamamen yazılımla tanımlanmış bir bilgisayar. Python'un sanal makinesi, bayt kodu derleyicisi tarafından yayınlanan *bytecode* 'u çalıştırır.

Python'un Zen'i Dili anlamaya ve kullanmaya yardımcı olan Python tasarım ilkeleri ve felsefelerinin listesi. Liste, etkileşimli komut isteminde `import this` yazarak bulunabilir.

Dokümanlar hakkında

Bu dokümanlar, Python dokümanları için özel olarak yazılmış bir doküman işlemcisi olan [Sphinx](#) tarafından [reStructuredText](#) kaynaklarından oluşturulur.

Dokümantasyonun ve araç zincirinin geliştirilmesi, tıpkı Python'un kendisi gibi tamamen gönüllü bir çabadır. Katkıda bulunmak istiyorsanız, nasıl yapacağınıza ilişkin bilgi için lütfen [reporting-bugs](#) sayfasına göz atın. Yeni gönüllülere her zaman açığız!

Destekleri için teşekkürler:

- Fred L. Drake, Jr., orijinal Python dokümantasyon araç setinin yaratıcısı ve içeriğin çoğunun yazarı;
- the [Docutils](#) project for creating reStructuredText and the Docutils suite;
- Fredrik Lundh for his Alternative Python Reference project from which Sphinx got many good ideas.

B.1 Python Dokümantasyonuna Katkıda Bulunanlar

Birçok kişi Python diline, Python standart kütüphanesine ve Python belgelerine katkıda bulunmuştur. Katkıda bulunanların kısmi listesi için Python kaynak dağıtımında [Misc/ACKS](#) adresine bakın.

Python topluluğunun girdileri ve katkılarıyla Python böyle harika bir dokümantasyona sahip – Teşekkürler!

Tarihçe ve Lisans

C.1 Yazılımın tarihçesi

Python, 1990'ların başında Guido van Rossum tarafından Hollanda'da Stichting Mathematisch Centrum'da (CWI, bkz. <https://www.cwi.nl/>) ABC adlı bir dilin devamı olarak oluşturuldu. Guido, diğerlerinin oldukça katkısı olmasına rağmen, Python'un ana yazarı olmaya devam ediyor.

1995'te Guido, yazılımın çeşitli sürümlerini yayınladığı Virginia, Reston'daki Ulusal Araştırma Girişimleri Kurumu'nda (CNRI, bkz. <https://www.cnri.reston.va.us/>) Python üzerindeki çalışmalarına devam etti.

Mayıs 2000'de, Guido ve Python çekirdek geliştirme ekibi, BeOpen PythonLabs ekibini oluşturmak için BeOpen.com'a taşındı. Aynı yılın Ekim ayında PythonLabs ekibi Digital Creations'a (şimdi Zope Corporation; bkz. <https://www.zope.org/>) taşındı. 2001 yılında, Python Yazılım Vakfı (PSF, bkz. <https://www.python.org/psf/>) kuruldu, özellikle Python ile ilgili Fikri Mülkiyete sahip olmak için oluşturulmuş kar amacı gütmeyen bir organizasyon. Zope Corporation, PSF'nin sponsor üyesidir.

Tüm Python sürümleri Açık Kaynaklıdır (Açık Kaynak Tanımı için bkz. <https://opensource.org/>). Tarihsel olarak, tümü olmasa da çoğu Python sürümleri de GPL uyumluydu; aşağıdaki tablo çeşitli yayınları özetlemektedir.

Yayın	Şundan türedi:	Yıl	Sahibi	GPL uyumlu mu?
0.9.0'dan 1.2'ye	n/a	1991-1995	CWI	evet
1.3 'dan 1.5.2'ye	1.2	1995-1999	CNRI	evet
1.6	1.5.2	2000	CNRI	hayır
2.0	1.6	2000	BeOpen.com	hayır
1.6.1	1.6	2001	CNRI	hayır
2.1	2.0+1.6.1	2001	PSF	hayır
2.0.1	2.0+1.6.1	2001	PSF	evet
2.1.1	2.1+2.0.1	2001	PSF	evet
2.1.2	2.1.1	2002	PSF	evet
2.1.3	2.1.2	2002	PSF	evet
2.2 ve üzeri	2.1.1	2001-Günümüz	PSF	evet

Not: GPL uyumlu olması, Python'u GPL kapsamında dağıttığımız anlamına gelmez. Tüm Python lisansları, GPL'den farklı olarak, değişikliklerinizi açık kaynak yapmadan değiştirilmiş bir sürümü dağıtmanıza izin verir. GPL uyumlu lisanslar, Python'u GPL kapsamında yayınlanan diğer yazılımlarla birleştirmeyi mümkün kılar; diğerleri yapmaz.

Bu yayınları mümkün kılmak için Guido'nun yönetimi altında çalışan birçok gönüllüye teşekkürler.

C.2 Python'a erişmek veya başka bir şekilde kullanmak için şartlar ve koşullar

Python yazılımı ve belgeleri *PSF Lisans Anlaşması* kapsamında lisanslanmıştır.

Python 3.8.6'dan başlayarak, belgelerdeki örnekler, tarifler ve diğer kodlar, PSF Lisans Sözleşmesi ve *Zero-Clause BSD license* kapsamında çift lisanslıdır.

Python'a dahil edilen bazı yazılımlar farklı lisanslar altındadır. Lisanslar, bu lisansa giren kodla listelenir. Bu lisansların eksik listesi için bkz. *Tüzel Yazılımlar için Lisanslar ve Onaylar*.

C.2.1 PYTHON İÇİN PSF LİSANS ANLAŞMASI 3.9.20

1. This LICENSE AGREEMENT is between the Python Software Foundation,
→ ("PSF"), and
the Individual or Organization ("Licensee") accessing and otherwise
→ using Python
3.9.20 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 3.9.20 alone or in any derivative
version, provided, however, that PSF's License Agreement and PSF's
→ notice of
copyright, i.e., "Copyright © 2001-2023 Python Software Foundation; All
→ Rights
Reserved" are retained in Python 3.9.20 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 3.9.20 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
3.9.20.
4. PSF is making Python 3.9.20 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 3.9.20 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.9.20

FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A
 ↳ RESULT OF
 MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.9.20, 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 3.9.20, Licensee agrees
 to be bound by the terms and conditions of this License Agreement.

C.2.2 PYTHON 2.0 İÇİN BEOPEN.COM LİSANS SÖZLEŞMESİ

BEOPEN PYTHON AÇIK KAYNAK LİSANS SÖZLEŞMESİ SÜRÜM 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.
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

(continues on next page)

(önceki sayfadan devam)

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 PYTHON 1.6.1 İÇİN CNRI LİSANS ANLAŞMASI

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.
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,

(continues on next page)

(önceki sayfadan devam)

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 0.9.0 ARASI 1.2 PYTHON İÇİN CWI LİSANS SÖZLEŞMESİ

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 PYTHON 3.9.20 BELGELERİNDEKİ KOD İÇİN SIFIR MADDE BSD LİSANSI

Permission to use, copy, modify, and/or distribute this software for any purpose with or without fee is hereby granted.

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 Tüzel Yazılımlar için Lisanslar ve Onaylar

Bu bölüm, Python dağıtımına dahil edilmiş üçüncü taraf yazılımlar için tamamlanmamış ancak büyüyen bir lisans ve onay listesidir.

C.3.1 Mersenne Twister'i

`_random` modülü, <http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html> adresinden indirilen kodu temel alır. Orijinal koddan kelimesi kelimesine yorumlar aşağıdadır:

```
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.

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 Soketler

The `socket` module uses the functions, `getaddrinfo()`, and `getnameinfo()`, which are coded in separate source files from the WIDE Project, <http://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
```

(continues on next page)

(önceki sayfadan devam)

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 Asenkron soket hizmetleri

asynchat ve asyncore modülleri aşağıdaki uyarıyı içerir:

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 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 Çerez yönetimi

http.cookies modülü aşağıdaki uyarıyı içerir:

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

(continues on next page)

(önceki sayfadan devam)

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 Çalıştırma izleme

trace modülü aşağıdaki uyarıyı içerir:

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 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 ve UUdecode fonksiyonları

uu modülü aşağıdaki uyarıyı içerir:

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

(continues on next page)

(önceki sayfadan devam)

```
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 Uzaktan Yordam Çağrıları

xmlrpc.client modülü aşağıdaki uyarıyı içerir:

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.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY 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

test_epoll modülü aşağıdaki uyarıyı içerir:

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

(continues on next page)

(önceki sayfadan devam)

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 kqueue seçin

select modülü, kqueue arayüzü için aşağıdaki uyarıyı içerir:

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 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

Python/pyhash.c dosyası, Dan Bernstein'in SipHash24 algoritmasının Marek Majkowski uygulamasını içerir. Burada aşağıdaki not yer alır:

<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>

(continues on next page)

(önceki sayfadan devam)

```
Original location:
  https://github.com/majek/csiphash/

Solution inspired by code from:
  Samuel Neves (supercop/crypto_auth/siphhash24/little)
  djb (supercop/crypto_auth/siphhash24/little2)
  Jean-Philippe Aumasson (https://131002.net/siphhash/siphhash24.c)
```

C.3.11 strtod ve 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 <http://www.netlib.org/fp/>. 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
 * OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
 *
 * *****/
```

C.3.12 OpenSSL

`hashlib`, `posix`, `ssl`, `crypt` modülleri, işletim sistemi tarafından sağlanmışsa ek performans için OpenSSL kütüphanesini kullanır. Ek olarak, Python için Windows ve macOS yükleyicileri, OpenSSL kütüphanelerinin bir kopyasını içerebilir, bu nedenle buraya OpenSSL lisansının bir kopyasını ekliyoruz:

```
LICENSE ISSUES
=====

The OpenSSL toolkit stays under a dual license, i.e. both the conditions of
the OpenSSL License and the original SSLeay license apply to the toolkit.
See below for the actual license texts. Actually both licenses are BSD-style
Open Source licenses. In case of any license issues related to OpenSSL
please contact openssl-core@openssl.org.

OpenSSL License
-----

/* =====
 * Copyright (c) 1998-2008 The OpenSSL 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:
```

(continues on next page)

(önceki sayfadan devam)

```

*
* 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. All advertising materials mentioning features or use of this
*    software must display the following acknowledgment:
*    "This product includes software developed by the OpenSSL Project
*    for use in the OpenSSL Toolkit. (http://www.openssl.org/)"
*
* 4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to
*    endorse or promote products derived from this software without
*    prior written permission. For written permission, please contact
*    openssl-core@openssl.org.
*
* 5. Products derived from this software may not be called "OpenSSL"
*    nor may "OpenSSL" appear in their names without prior written
*    permission of the OpenSSL Project.
*
* 6. Redistributions of any form whatsoever must retain the following
*    acknowledgment:
*    "This product includes software developed by the OpenSSL Project
*    for use in the OpenSSL Toolkit (http://www.openssl.org/)"
*
* THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ``AS IS'' AND ANY
* EXPRESSED 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 OpenSSL PROJECT OR
* ITS 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.
* =====
*
* This product includes cryptographic software written by Eric Young
* (eay@cryptsoft.com). This product includes software written by Tim
* Hudson (tjh@cryptsoft.com).
*
*/

```

Original SSLeay License

```

/* Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com)
* All rights reserved.
*
* This package is an SSL implementation written
* by Eric Young (eay@cryptsoft.com).
* The implementation was written so as to conform with Netscapes SSL.
*
* This library is free for commercial and non-commercial use as long as
* the following conditions are aheared to. The following conditions
* apply to all code found in this distribution, be it the RC4, RSA,

```

(continues on next page)

(önceki sayfadan devam)

```

* lhash, DES, etc., code; not just the SSL code. The SSL documentation
* included with this distribution is covered by the same copyright terms
* except that the holder is Tim Hudson (tjh@cryptsoft.com).
*
* Copyright remains Eric Young's, and as such any Copyright notices in
* the code are not to be removed.
* If this package is used in a product, Eric Young should be given attribution
* as the author of the parts of the library used.
* This can be in the form of a textual message at program startup or
* in documentation (online or textual) provided with the package.
*
* 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 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. All advertising materials mentioning features or use of this software
* must display the following acknowledgement:
* "This product includes cryptographic software written by
* Eric Young (eay@cryptsoft.com)"
* The word 'cryptographic' can be left out if the routines from the library
* being used are not cryptographic related :-).
* 4. If you include any Windows specific code (or a derivative thereof) from
* the apps directory (application code) you must include an acknowledgement:
* "This product includes software written by Tim Hudson (tjh@cryptsoft.com)"
*
* THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``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
* SUCH DAMAGE.
*
* The licence and distribution terms for any publically available version or
* derivative of this code cannot be changed. i.e. this code cannot simply be
* copied and put under another distribution licence
* [including the GNU Public Licence.]
*/

```

C.3.13 expat

pyexpat uzantısı, derleme --with-system-expat şeklinde yapılandırılmadığı sürece, expat kaynaklarının dahil edildiği bir kopya kullanılarak oluşturulur:

```

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,

```

(continues on next page)

(önceki sayfadan devam)

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

`_ctypes` uzantısı, yapı `--with-system-libffi` olarak yapılandırılmadığı sürece libffi kaynaklarının dahil edildiği bir kopya kullanılarak oluşturulur:

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 ``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

`zlib` uzantısı, sistemde bulunan `zlib` sürümü derleme için kullanılamayacak kadar eskiyse, `zlib` kaynaklarının dahil edildiği bir kopya kullanılarak oluşturulur:

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:

(continues on next page)

(önceki sayfadan devam)

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
jloup@gzip.org

Mark Adler
madler@alumni.caltech.edu

C.3.16 cfuhash

tracemalloc tarafından kullanılan hash tablosunun uygulanması cfuhash projesine dayanmaktadır:

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 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

`_decimal` modülü, `yapı --with-system-libmpdec` şeklinde yapılandırılmadığı sürece libmpdec kitaplığının dahil edildiği bir kopya kullanılarak oluşturulur:

```
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  
SUCH DAMAGE.
```

C.3.18 W3C C14N test paketi

`test` paketindeki C14N 2.0 test paketi (`Lib/test/xmltestdata/c14n-20/`), <https://www.w3.org/TR/xml-c14n2-testcases/> adresindeki W3C web sitesinden alınmıştır ve 3 maddeli BSD lisansı altında dağıtılmaktadır:

```
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
```

(continues on next page)

(önceki sayfadan devam)

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.

Telif Hakkı

Python ve bu dokümantasyon:

Telif Hakkı © 2001-2023 Python Software Foundation. Tüm hakları saklıdır.

Telif Hakkı © 2000 BeOpen.com. Tüm hakları saklıdır.

Telif Hakkı © 1995-2000 Ulusal Araştırma Girişimleri Kurumu. Tüm hakları saklıdır.

Telif Hakkı © 1991-1995 Stichting Mathematisch Centrum. Tüm hakları saklıdır.

Bütün lisans ve izin bilgileri için [Tarihçe ve Lisans](#) 'a göz atın.

Non-alphabetical

..., [225](#)
 2to3, [225](#)
 >>>, [225](#)
 __all__ (*package variable*), [40](#)
 __dict__ (*module attribute*), [121](#)
 __doc__ (*module attribute*), [121](#)
 __file__ (*module attribute*), [121](#), [122](#)
 __future__, [229](#)
 __import__
 gömülü fonksiyon, [40](#)
 __loader__ (*module attribute*), [121](#)
 __main__
 modülü, [11](#), [142](#), [152](#)
 __name__ (*module attribute*), [121](#)
 __package__ (*module attribute*), [121](#)
 __slots__, [235](#)
 _frozen (*C tipi*), [42](#)
 _inittab (*C tipi*), [42](#)
 _Py_c_diff (*C fonksiyonu*), [87](#)
 _Py_c_neg (*C fonksiyonu*), [87](#)
 _Py_c_pow (*C fonksiyonu*), [87](#)
 _Py_c_prod (*C fonksiyonu*), [87](#)
 _Py_c_quot (*C fonksiyonu*), [87](#)
 _Py_c_sum (*C fonksiyonu*), [87](#)
 _Py_InitializeMain (*C fonksiyonu*), [173](#)
 _Py_NoneStruct (*C değişkeni*), [183](#)
 _PyBytes_Resize (*C fonksiyonu*), [90](#)
 _PyCFunctionFast (*C tipi*), [185](#)
 _PyCFunctionFastWithKeywords (*C tipi*), [185](#)
 _PyFrameEvalFunction (*C tipi*), [151](#)
 _PyInterpreterState_GetEvalFrameFunc (*C fonksiyonu*), [151](#)
 _PyInterpreterState_SetEvalFrameFunc (*C fonksiyonu*), [151](#)
 _PyObject_New (*C fonksiyonu*), [183](#)
 _PyObject_NewVar (*C fonksiyonu*), [183](#)
 _PyTuple_Resize (*C fonksiyonu*), [110](#)
 _thread
 modülü, [148](#)

A

abort(), [39](#)
 abs

 gömülü fonksiyon, [66](#)
 ad alanı, [232](#)
 ad alanı paketi, [233](#)
 adlandırılmış demet, [232](#)
 allocfunc (*C tipi*), [217](#)
 anahtar işlev, [231](#)
 anahtar kelime argümanı, [231](#)
 anlamak, [235](#)
 argüman, [225](#)
 argv (*in module sys*), [145](#)
 ascii
 gömülü fonksiyon, [59](#)
 asenkron bağlam yöneticisi, [226](#)
 asenkron jeneratör, [226](#)
 asenkron jeneratör yineleyici, [226](#)
 asenkron yineleyici, [226](#)

B

bağlam değişkeni, [227](#)
 bağlam yöneticisi, [227](#)
 bayt benzeri nesne, [226](#)
 bayt kodu, [227](#)
 BDFL, [226](#)
 beklenebilir, [226](#)
 belge dizisi, [228](#)
 binaryfunc (*C tipi*), [218](#)
 bitişik, [227](#)
 BOŞTA, [230](#)
 buffer interface
 (see buffer protocol), [71](#)
 buffer object
 (see buffer protocol), [71](#)
 buffer protocol, [71](#)
 builtins
 modülü, [11](#), [142](#), [152](#)
 bulucu, [229](#)
 bytearray
 nesne, [90](#)
 bytes
 gömülü fonksiyon, [59](#)
 nesne, [88](#)

C

calloc(), [175](#)

Capsule
 nesne, 131
C-contiguous, 74, 227
classmethod
 gömülü fonksiyon, 187
cleanup functions, 39
close() (in module os), 153
CO_FUTURE_DIVISION (C değişkeni), 19
code object, 119
compile
 gömülü fonksiyon, 41
complex number
 nesne, 87
contiguous, 74
copyright (in module sys), 144
CPython, 227
create_module (C fonksiyonu), 124

Ç

çevre değişkeni
 exec_prefix, 4
 PATH, 11
 prefix, 4
 PYTHON*, 140
 PYTHONCOERCECLOCALE, 170
 PYTHONDEBUG, 140
 PYTHONDONTWRITEBYTECODE, 140
 PYTHONDUMPPREFS, 195
 PYTHONHASHSEED, 140
 PYTHONHOME, 11, 140, 145, 166
 PYTHONINSPECT, 141
 PYTHONIOENCODING, 143
 PYTHONLEGACYWINDOWSFSENCODING, 141
 PYTHONLEGACYWINDOWSTDIO, 141
 PYTHONMALLOC, 176, 179, 180
 PYTHONMALLOCSTATS, 176
 PYTHONNOUSERSITE, 141
 PYTHONOLDPARSER, 168
 PYTHONOPTIMIZE, 141
 PYTHONPATH, 11, 140, 166
 PYTHONUNBUFFERED, 141
 PYTHONUTF8, 170
 PYTHONVERBOSE, 141
çöp toplama, 229

D

değişken açıklama, 236
değişmez, 230
değiştirilebilir, 232
dekoratör, 227
descrgetfunc (C tipi), 217
descrsetfunc (C tipi), 217
destructor (C tipi), 217
dictionary
 nesne, 113
dipnot, 225
divmod
 gömülü fonksiyon, 66

dizi, 235
dosya benzeri nesne, 229
dosya nesnesi, 228
duck-typing, 228

E

EAFP, 228
EOFError (built-in exception), 120
eşyordam, 227
eşyordam işlevi, 227
eşzamansız yinelenenbilir, 226
etkileşimli, 230
evrensel yeni satırlar, 236
exc_info() (in module sys), 9
exec_module (C fonksiyonu), 125
exec_prefix, 4
executable (in module sys), 144
exit(), 39

F

f-string, 228
file
 nesne, 120
float
 gömülü fonksiyon, 67
floating point
 nesne, 86
fonksiyon, 229
fonksiyon açıklaması, 229
Fortran contiguous, 74, 227
free(), 175
freefunc (C tipi), 217
freeze utility, 42
frozenset
 nesne, 115
function
 nesne, 117

G

geçici API, 234
geçici paket, 234
genel işlev, 229
genel tercüman kilidi, 230
genel tip, 230
generator, 229
generator expression, 229
geri çağırmak, 227
getattrfunc (C tipi), 217
getattrofunc (C tipi), 217
getbufferproc (C tipi), 218
getiterfunc (C tipi), 217
GIL, 230
global interpreter lock, 146
gömülü fonksiyon
 __import__, 40
 abs, 66
 ascii, 59
 bytes, 59

classmethod, 187
 compile, 41
 divmod, 66
 float, 67
 hash, 59, 199
 int, 67
 len, 60, 68, 70, 112, 114, 116
 pow, 66, 67
 repr, 59, 199
 staticmethod, 187
 tuple, 69, 113
 type, 60

H

haritalama, 232
 hash
 gömülü fonksiyon, 59, 199
 hashfunc (*C tipi*), 217

I

iç içe kapsam, 233
 içe aktarıcı, 230
 içe aktarım yolu, 230
 içe aktarma, 230
 ifade (*değer döndürmez*), 235
 ifade (*değer döndürür*), 228
 ikili dosya, 226
 incr_item(), 10, 11
 initproc (*C tipi*), 217
 inquiry (*C tipi*), 222
 instancemethod
 nesne, 118
 int
 gömülü fonksiyon, 67
 integer
 nesne, 83
 interpreter lock, 146
 iternextfunc (*C tipi*), 218

J

jeneratör, 229
 jeneratör ifadesi, 229
 jeneratör yineleyici, 229

K

karma tabanlı pyc, 230
 karmaşık sayı, 227
 kat bölümü, 229
 KeyboardInterrupt (*built-in exception*), 29
 kısım, 234
 konumsal argüman, 234

L

lambda, 231
 LBYL, 231
 len
 gömülü fonksiyon, 60, 68, 70, 112, 114, 116

lenfunc (*C tipi*), 218
 list
 nesne, 112
 liste, 231
 liste anlama, 231
 lock, interpreter, 146
 long integer
 nesne, 83
 LONG_MAX, 84

M

magic
 method, 232
 main(), 143, 145
 malloc(), 175
 mapping
 nesne, 113
 memoryview
 nesne, 129
 meta yol bulucu, 232
 metasinif, 232
 METH_CLASS (*gömülü değişken*), 187
 METH_COEXIST (*gömülü değişken*), 187
 METH_FASTCALL (*gömülü değişken*), 186
 METH_NOARGS (*gömülü değişken*), 187
 METH_O (*gömülü değişken*), 187
 METH_STATIC (*gömülü değişken*), 187
 METH_VARARGS (*gömülü değişken*), 186
 method
 magic, 232
 nesne, 118
 special, 235
 MethodType (*in module types*), 117, 118
 metot, 232
 metot kalite sıralaması, 232
 module
 nesne, 121
 search path, 11, 142, 144
 modules (*in module sys*), 40, 142
 ModuleType (*in module types*), 121
 modül, 232
 modül özelliği, 232
 modülü
 __main__, 11, 142, 152
 _thread, 148
 builtins, 11, 142, 152
 signal, 29
 sys, 11, 142, 152
 MRO, 232

N

nesne
 bytearray, 90
 bytes, 88
 Capsule, 131
 complex number, 87
 dictionary, 113
 file, 120

- floating point, 86
- frozenset, 115
- function, 117
- instancemethod, 118
- integer, 83
- list, 112
- long integer, 83
- mapping, 113
- memoryview, 129
- method, 118
- module, 121
- None, 82
- numeric, 83
- sequence, 88
- set, 115
- tuple, 109
- type, 6, 79
- newfunc (*C tipi*), 217
- nitelik, 226
- nitelikli isim, 234
- None
 - nesne, 82
- numeric
 - nesne, 83

O

- obje, 233
- object
 - code, 119
- objobjargproc (*C tipi*), 218
- objobjproc (*C tipi*), 218
- OverflowError (*built-in exception*), 84, 85

Ö

- özel metod, 235

P

- package variable
 - __all__, 40
- paket, 233
- parametre, 233
- parçalamak, 235
- PATH, 11
- path
 - module search, 11, 142, 144
- path (*in module sys*), 11, 142, 144
- PEP, 234
- platform (*in module sys*), 144
- pow
 - gömülü fonksiyon, 66, 67
- prefix, 4
- Py_ABS (*C makrosu*), 4
- Py_AddPendingCall (*C fonksiyonu*), 154
- Py_AddPendingCall(), 154
- Py_AtExit (*C fonksiyonu*), 39
- Py_BEGIN_ALLOW_THREADS, 146
- Py_BEGIN_ALLOW_THREADS (*C makrosu*), 149
- Py_BLOCK_THREADS (*C makrosu*), 149

- Py_buffer (*C tipi*), 72
- Py_buffer.buf (*C ögesi*), 72
- Py_buffer.format (*C ögesi*), 73
- Py_buffer.internal (*C ögesi*), 73
- Py_buffer.itemsize (*C ögesi*), 72
- Py_buffer.len (*C ögesi*), 72
- Py_buffer.ndim (*C ögesi*), 73
- Py_buffer.obj (*C ögesi*), 72
- Py_buffer.readonly (*C ögesi*), 72
- Py_buffer.shape (*C ögesi*), 73
- Py_buffer.strides (*C ögesi*), 73
- Py_buffer.suboffsets (*C ögesi*), 73
- Py_BuildValue (*C fonksiyonu*), 49
- Py_BytesMain (*C fonksiyonu*), 15
- Py_BytesWarningFlag (*C değişkeni*), 140
- Py_CHARMASK (*C makrosu*), 5
- Py_CLEAR (*C fonksiyonu*), 21
- Py_CompileString (*C fonksiyonu*), 18
- Py_CompileString(), 19
- Py_CompileStringExFlags (*C fonksiyonu*), 18
- Py_CompileStringFlags (*C fonksiyonu*), 18
- Py_CompileStringObject (*C fonksiyonu*), 18
- Py_complex (*C tipi*), 87
- Py_DebugFlag (*C değişkeni*), 140
- Py_DecodeLocale (*C fonksiyonu*), 36
- Py_DECREF (*C fonksiyonu*), 21
- Py_DECREF(), 6
- Py_DEPRECATED (*C makrosu*), 5
- Py_DontWriteBytecodeFlag (*C değişkeni*), 140
- Py_Ellipsis (*C değişkeni*), 129
- Py_EncodeLocale (*C fonksiyonu*), 37
- Py_END_ALLOW_THREADS, 146
- Py_END_ALLOW_THREADS (*C makrosu*), 149
- Py_EndInterpreter (*C fonksiyonu*), 153
- Py_EnterRecursiveCall (*C fonksiyonu*), 31
- Py_eval_input (*C değişkeni*), 19
- Py_Exit (*C fonksiyonu*), 39
- Py_ExitStatusException (*C fonksiyonu*), 161
- Py_False (*C değişkeni*), 86
- Py_FatalError (*C fonksiyonu*), 39
- Py_FatalError(), 145
- Py_FdIsInteractive (*C fonksiyonu*), 35
- Py_file_input (*C değişkeni*), 19
- Py_Finalize (*C fonksiyonu*), 142
- Py_FinalizeEx (*C fonksiyonu*), 142
- Py_FinalizeEx(), 39, 142, 153
- Py_FrozenFlag (*C değişkeni*), 140
- Py_GenericAlias (*C fonksiyonu*), 137
- Py_GenericAliasType (*C değişkeni*), 137
- Py_GetArgcArgv (*C fonksiyonu*), 172
- Py_GetBuildInfo (*C fonksiyonu*), 145
- Py_GetCompiler (*C fonksiyonu*), 144
- Py_GetCopyright (*C fonksiyonu*), 144
- Py_GETENV (*C makrosu*), 5
- Py_GetExecPrefix (*C fonksiyonu*), 143
- Py_GetExecPrefix(), 11
- Py_GetPath (*C fonksiyonu*), 144
- Py_GetPath(), 11, 143, 144

- Py_GetPlatform (*C fonksiyonu*), 144
Py_GetPrefix (*C fonksiyonu*), 143
Py_GetPrefix (), 11
Py_GetProgramFullPath (*C fonksiyonu*), 144
Py_GetProgramFullPath (), 11
Py_GetProgramName (*C fonksiyonu*), 143
Py_GetPythonHome (*C fonksiyonu*), 145
Py_GetVersion (*C fonksiyonu*), 144
Py_HashRandomizationFlag (*C değişkeni*), 140
Py_IgnoreEnvironmentFlag (*C değişkeni*), 140
Py_INCREF (*C fonksiyonu*), 21
Py_INCREF (), 6
Py_Initialize (*C fonksiyonu*), 142
Py_Initialize (), 11, 143, 153
Py_InitializeEx (*C fonksiyonu*), 142
Py_InitializeFromConfig (*C fonksiyonu*), 168
Py_InspectFlag (*C değişkeni*), 140
Py_InteractiveFlag (*C değişkeni*), 141
Py_IS_TYPE (*C fonksiyonu*), 184
Py_IsInitialized (*C fonksiyonu*), 142
Py_IsInitialized (), 11
Py_IsolatedFlag (*C değişkeni*), 141
Py_LeaveRecursiveCall (*C fonksiyonu*), 31
Py_LegacyWindowsFSEncodingFlag (*C değişkeni*), 141
Py_LegacyWindowsStdioFlag (*C değişkeni*), 141
Py_Main (*C fonksiyonu*), 15
Py_MAX (*C makrosu*), 4
Py_MEMBER_SIZE (*C makrosu*), 5
Py_MIN (*C makrosu*), 4
Py_mod_create (*C makrosu*), 124
Py_mod_exec (*C makrosu*), 125
Py_NewInterpreter (*C fonksiyonu*), 152
Py_None (*C değişkeni*), 82
Py_NoSiteFlag (*C değişkeni*), 141
Py_NotImplemented (*C değişkeni*), 57
Py_NoUserSiteDirectory (*C değişkeni*), 141
Py_OptimizeFlag (*C değişkeni*), 141
Py_PreInitialize (*C fonksiyonu*), 163
Py_PreInitializeFromArgs (*C fonksiyonu*), 163
Py_PreInitializeFromBytesArgs (*C fonksiyonu*), 163
Py_PRINT_RAW, 121
Py_QuietFlag (*C değişkeni*), 141
Py_REFCNT (*C makrosu*), 184
Py_ReprEnter (*C fonksiyonu*), 31
Py_ReprLeave (*C fonksiyonu*), 32
Py_RETURN_FALSE (*C makrosu*), 86
Py_RETURN_NONE (*C makrosu*), 82
Py_RETURN_NOTIMPLEMENTED (*C makrosu*), 57
Py_RETURN_RICHCOMPARE (*C makrosu*), 205
Py_RETURN_TRUE (*C makrosu*), 86
Py_RunMain (*C fonksiyonu*), 172
Py_SET_REFCNT (*C fonksiyonu*), 185
Py_SET_SIZE (*C fonksiyonu*), 185
Py_SET_TYPE (*C fonksiyonu*), 184
Py_SetPath (), 144
Py_SetProgramName (*C fonksiyonu*), 143
Py_SetProgramName (), 11, 142, 144
Py_SetPythonHome (*C fonksiyonu*), 145
Py_SetStandardStreamEncoding (*C fonksiyonu*), 143
Py_SIZE (*C makrosu*), 185
Py_single_input (*C değişkeni*), 19
PY_SSIZE_T_MAX, 84
Py_ssize_t (*C tipi*), 9
Py_STRINGIFY (*C makrosu*), 5
Py_TPFLAGS_BASE_EXC_SUBCLASS (*gömülü değişken*), 202
Py_TPFLAGS_BASETYPE (*gömülü değişken*), 201
Py_TPFLAGS_BYTES_SUBCLASS (*gömülü değişken*), 202
Py_TPFLAGS_DEFAULT (*gömülü değişken*), 202
Py_TPFLAGS_DICT_SUBCLASS (*gömülü değişken*), 202
Py_TPFLAGS_HAVE_FINALIZE (*gömülü değişken*), 202
Py_TPFLAGS_HAVE_GC (*gömülü değişken*), 202
Py_TPFLAGS_HAVE_VECTORCALL (*gömülü değişken*), 203
Py_TPFLAGS_HEAPTYPE (*gömülü değişken*), 201
Py_TPFLAGS_LIST_SUBCLASS (*gömülü değişken*), 202
Py_TPFLAGS_LONG_SUBCLASS (*gömülü değişken*), 202
Py_TPFLAGS_METHOD_DESCRIPTOR (*gömülü değişken*), 202
Py_TPFLAGS_READY (*gömülü değişken*), 201
Py_TPFLAGS_READYING (*gömülü değişken*), 201
Py_TPFLAGS_TUPLE_SUBCLASS (*gömülü değişken*), 202
Py_TPFLAGS_TYPE_SUBCLASS (*gömülü değişken*), 202
Py_TPFLAGS_UNICODE_SUBCLASS (*gömülü değişken*), 202
Py_tracefunc (*C tipi*), 154
Py_True (*C değişkeni*), 86
Py_tss_NEEDS_INIT (*C makrosu*), 156
Py_tss_t (*C tipi*), 156
Py_TYPE (*C makrosu*), 184
Py_UCS1 (*C tipi*), 91
Py_UCS2 (*C tipi*), 91
Py_UCS4 (*C tipi*), 91
Py_UNBLOCK_THREADS (*C makrosu*), 149
Py_UnbufferedStdioFlag (*C değişkeni*), 141
Py_UNICODE (*C tipi*), 91
Py_UNICODE_IS_HIGH_SURROGATE (*C makrosu*), 95
Py_UNICODE_IS_LOW_SURROGATE (*C makrosu*), 95
Py_UNICODE_IS_SURROGATE (*C makrosu*), 95
Py_UNICODE_ISALNUM (*C fonksiyonu*), 94
Py_UNICODE_ISALPHA (*C fonksiyonu*), 94
Py_UNICODE_ISDECIMAL (*C fonksiyonu*), 94

- Py_UNICODE_ISDIGIT (*C fonksiyonu*), 94
- Py_UNICODE_ISLINEBREAK (*C fonksiyonu*), 94
- Py_UNICODE_ISLOWER (*C fonksiyonu*), 94
- Py_UNICODE_ISNUMERIC (*C fonksiyonu*), 94
- Py_UNICODE_ISPRINTABLE (*C fonksiyonu*), 94
- Py_UNICODE_ISSPACE (*C fonksiyonu*), 94
- Py_UNICODE_ISTITLE (*C fonksiyonu*), 94
- Py_UNICODE_ISUPPER (*C fonksiyonu*), 94
- Py_UNICODE_JOIN_SURROGATES (*C makrosu*), 95
- Py_UNICODE_TODECIMAL (*C fonksiyonu*), 94
- Py_UNICODE_TODIGIT (*C fonksiyonu*), 94
- Py_UNICODE_TOLOWER (*C fonksiyonu*), 94
- Py_UNICODE_TONUMERIC (*C fonksiyonu*), 94
- Py_UNICODE_TOTITLE (*C fonksiyonu*), 94
- Py_UNICODE_TOUPPER (*C fonksiyonu*), 94
- Py_UNREACHABLE (*C makrosu*), 4
- Py_UNUSED (*C makrosu*), 5
- Py_VaBuildValue (*C fonksiyonu*), 51
- PY_VECTORCALL_ARGUMENTS_OFFSET (*C makrosu*), 61
- Py_VerboseFlag (*C değişkeni*), 141
- Py_VISIT (*C fonksiyonu*), 222
- Py_XDECREF (*C fonksiyonu*), 21
- Py_XDECREF (), 11
- Py_XINCREf (*C fonksiyonu*), 21
- PyAnySet_Check (*C fonksiyonu*), 116
- PyAnySet_CheckExact (*C fonksiyonu*), 116
- PyArg_Parse (*C fonksiyonu*), 49
- PyArg_ParseTuple (*C fonksiyonu*), 48
- PyArg_ParseTupleAndKeywords (*C fonksiyonu*), 48
- PyArg_UnpackTuple (*C fonksiyonu*), 49
- PyArg_ValidateKeywordArguments (*C fonksiyonu*), 49
- PyArg_VaParse (*C fonksiyonu*), 48
- PyArg_VaParseTupleAndKeywords (*C fonksiyonu*), 48
- PyASCIIObject (*C tipi*), 91
- PyAsyncMethods (*C tipi*), 216
- PyAsyncMethods.am_aiter (*C ögesi*), 216
- PyAsyncMethods.am_anext (*C ögesi*), 217
- PyAsyncMethods.am_await (*C ögesi*), 216
- PyBool_Check (*C fonksiyonu*), 86
- PyBool_FromLong (*C fonksiyonu*), 86
- PyBUF_ANY_CONTIGUOUS (*C makrosu*), 75
- PyBUF_C_CONTIGUOUS (*C makrosu*), 75
- PyBUF_CONTIG (*C makrosu*), 75
- PyBUF_CONTIG_RO (*C makrosu*), 75
- PyBUF_F_CONTIGUOUS (*C makrosu*), 75
- PyBUF_FORMAT (*C makrosu*), 74
- PyBUF_FULL (*C makrosu*), 75
- PyBUF_FULL_RO (*C makrosu*), 75
- PyBUF_INDIRECT (*C makrosu*), 74
- PyBUF_ND (*C makrosu*), 74
- PyBUF_RECORDS (*C makrosu*), 75
- PyBUF_RECORDS_RO (*C makrosu*), 75
- PyBUF_SIMPLE (*C makrosu*), 74
- PyBUF_STRIDED (*C makrosu*), 75
- PyBUF_STRIDED_RO (*C makrosu*), 75
- PyBUF_STRIDES (*C makrosu*), 74
- PyBUF_WRITABLE (*C makrosu*), 74
- PyBuffer_FillContiguousStrides (*C fonksiyonu*), 77
- PyBuffer_FillInfo (*C fonksiyonu*), 77
- PyBuffer_FromContiguous (*C fonksiyonu*), 77
- PyBuffer_GetPointer (*C fonksiyonu*), 77
- PyBuffer_IsContiguous (*C fonksiyonu*), 77
- PyBuffer_Release (*C fonksiyonu*), 77
- PyBuffer_SizeFromFormat (*C fonksiyonu*), 77
- PyBuffer_ToContiguous (*C fonksiyonu*), 77
- PyBufferProcs, 71
- PyBufferProcs (*C tipi*), 215
- PyBufferProcs.bf_getbuffer (*C ögesi*), 215
- PyBufferProcs.bf_releasebuffer (*C ögesi*), 216
- PyByteArray_AS_STRING (*C fonksiyonu*), 91
- PyByteArray_AsString (*C fonksiyonu*), 90
- PyByteArray_Check (*C fonksiyonu*), 90
- PyByteArray_CheckExact (*C fonksiyonu*), 90
- PyByteArray_Concat (*C fonksiyonu*), 90
- PyByteArray_FromObject (*C fonksiyonu*), 90
- PyByteArray_FromStringAndSize (*C fonksiyonu*), 90
- PyByteArray_GET_SIZE (*C fonksiyonu*), 91
- PyByteArray_Resize (*C fonksiyonu*), 90
- PyByteArray_Size (*C fonksiyonu*), 90
- PyByteArray_Type (*C değişkeni*), 90
- PyByteArrayObject (*C tipi*), 90
- PyBytes_AS_STRING (*C fonksiyonu*), 89
- PyBytes_AsString (*C fonksiyonu*), 89
- PyBytes_AsStringAndSize (*C fonksiyonu*), 89
- PyBytes_Check (*C fonksiyonu*), 88
- PyBytes_CheckExact (*C fonksiyonu*), 88
- PyBytes_Concat (*C fonksiyonu*), 89
- PyBytes_ConcatAndDel (*C fonksiyonu*), 90
- PyBytes_FromFormat (*C fonksiyonu*), 88
- PyBytes_FromFormatV (*C fonksiyonu*), 89
- PyBytes_FromObject (*C fonksiyonu*), 89
- PyBytes_FromString (*C fonksiyonu*), 88
- PyBytes_FromStringAndSize (*C fonksiyonu*), 88
- PyBytes_GET_SIZE (*C fonksiyonu*), 89
- PyBytes_Size (*C fonksiyonu*), 89
- PyBytes_Type (*C değişkeni*), 88
- PyBytesObject (*C tipi*), 88
- PyCallable_Check (*C fonksiyonu*), 65
- PyCallIter_Check (*C fonksiyonu*), 127
- PyCallIter_New (*C fonksiyonu*), 127
- PyCallIter_Type (*C değişkeni*), 127
- PyCapsule (*C tipi*), 131
- PyCapsule_CheckExact (*C fonksiyonu*), 131
- PyCapsule_Destructor (*C tipi*), 131
- PyCapsule_GetContext (*C fonksiyonu*), 131
- PyCapsule_GetDestructor (*C fonksiyonu*), 131
- PyCapsule_GetName (*C fonksiyonu*), 131
- PyCapsule_GetPointer (*C fonksiyonu*), 131

- PyCapsule_Import (*C fonksiyonu*), 131
- PyCapsule_IsValid (*C fonksiyonu*), 132
- PyCapsule_New (*C fonksiyonu*), 131
- PyCapsule_SetContext (*C fonksiyonu*), 132
- PyCapsule_SetDestructor (*C fonksiyonu*), 132
- PyCapsule_SetName (*C fonksiyonu*), 132
- PyCapsule_SetPointer (*C fonksiyonu*), 132
- PyCell_Check (*C fonksiyonu*), 119
- PyCell_GET (*C fonksiyonu*), 119
- PyCell_Get (*C fonksiyonu*), 119
- PyCell_New (*C fonksiyonu*), 119
- PyCell_SET (*C fonksiyonu*), 119
- PyCell_Set (*C fonksiyonu*), 119
- PyCell_Type (*C değişkeni*), 119
- PyCellobject (*C tipi*), 119
- PyCFunction (*C tipi*), 185
- PyCFunctionWithKeywords (*C tipi*), 185
- PyCMethod (*C tipi*), 186
- PyCode_Check (*C fonksiyonu*), 119
- PyCode_GetNumFree (*C fonksiyonu*), 119
- PyCode_New (*C fonksiyonu*), 119
- PyCode_NewEmpty (*C fonksiyonu*), 120
- PyCode_NewWithPosOnlyArgs (*C fonksiyonu*), 120
- PyCode_Type (*C değişkeni*), 119
- PyCodec_BackslashReplaceErrors (*C fonksiyonu*), 55
- PyCodec_Decompile (*C fonksiyonu*), 54
- PyCodec_Decoder (*C fonksiyonu*), 54
- PyCodec_Encode (*C fonksiyonu*), 53
- PyCodec_Encoder (*C fonksiyonu*), 54
- PyCodec_IgnoreErrors (*C fonksiyonu*), 54
- PyCodec_IncrementalDecoder (*C fonksiyonu*), 54
- PyCodec_IncrementalEncoder (*C fonksiyonu*), 54
- PyCodec_KnownEncoding (*C fonksiyonu*), 53
- PyCodec_LookupError (*C fonksiyonu*), 54
- PyCodec_NameReplaceErrors (*C fonksiyonu*), 55
- PyCodec_Register (*C fonksiyonu*), 53
- PyCodec_RegisterError (*C fonksiyonu*), 54
- PyCodec_ReplaceErrors (*C fonksiyonu*), 54
- PyCodec_StreamReader (*C fonksiyonu*), 54
- PyCodec_StreamWriter (*C fonksiyonu*), 54
- PyCodec_StrictErrors (*C fonksiyonu*), 54
- PyCodec_XMLCharRefReplaceErrors (*C fonksiyonu*), 55
- PyCodeObject (*C tipi*), 119
- PyCompactUnicodeObject (*C tipi*), 91
- PyCompilerFlags (*C tipi*), 19
- PyCompilerFlags.cf_feature_version (*C ögesi*), 19
- PyCompilerFlags.cf_flags (*C ögesi*), 19
- PyComplex_AsCComplex (*C fonksiyonu*), 88
- PyComplex_Check (*C fonksiyonu*), 87
- PyComplex_CheckExact (*C fonksiyonu*), 87
- PyComplex_FromCComplex (*C fonksiyonu*), 88
- PyComplex_FromDoubles (*C fonksiyonu*), 88
- PyComplex_ImagAsDouble (*C fonksiyonu*), 88
- PyComplex_RealAsDouble (*C fonksiyonu*), 88
- PyComplex_Type (*C değişkeni*), 87
- PyComplexObject (*C tipi*), 87
- PyConfig (*C tipi*), 164
- PyConfig_Clear (*C fonksiyonu*), 164
- PyConfig_InitIsolatedConfig (*C fonksiyonu*), 164
- PyConfig_InitPythonConfig (*C fonksiyonu*), 164
- PyConfig_Read (*C fonksiyonu*), 164
- PyConfig_SetArgv (*C fonksiyonu*), 164
- PyConfig_SetBytesArgv (*C fonksiyonu*), 164
- PyConfig_SetBytesString (*C fonksiyonu*), 164
- PyConfig_SetString (*C fonksiyonu*), 164
- PyConfig_SetWideStringList (*C fonksiyonu*), 164
- PyConfig._use_peg_parser (*C ögesi*), 168
- PyConfig.argv (*C ögesi*), 165
- PyConfig.base_exec_prefix (*C ögesi*), 165
- PyConfig.base_executable (*C ögesi*), 165
- PyConfig.base_prefix (*C ögesi*), 165
- PyConfig.buffered_stdio (*C ögesi*), 165
- PyConfig.bytes_warning (*C ögesi*), 165
- PyConfig.check_hash_pycs_mode (*C ögesi*), 165
- PyConfig.configure_c_stdio (*C ögesi*), 165
- PyConfig.dev_mode (*C ögesi*), 165
- PyConfig.dump_refs (*C ögesi*), 165
- PyConfig.exec_prefix (*C ögesi*), 165
- PyConfig.executable (*C ögesi*), 165
- PyConfig.fault_handler (*C ögesi*), 165
- PyConfig.filesystem_encoding (*C ögesi*), 165
- PyConfig.filesystem_errors (*C ögesi*), 166
- PyConfig.hash_seed (*C ögesi*), 166
- PyConfig.home (*C ögesi*), 166
- PyConfig.import_time (*C ögesi*), 166
- PyConfig.inspect (*C ögesi*), 166
- PyConfig.install_signal_handlers (*C ögesi*), 166
- PyConfig.interactive (*C ögesi*), 166
- PyConfig.isolated (*C ögesi*), 166
- PyConfig.legacy_windows_stdio (*C ögesi*), 166
- PyConfig.malloc_stats (*C ögesi*), 166
- PyConfig.module_search_paths (*C ögesi*), 166
- PyConfig.module_search_paths_set (*C ögesi*), 166
- PyConfig.optimization_level (*C ögesi*), 166
- PyConfig.parse_argv (*C ögesi*), 166
- PyConfig.parser_debug (*C ögesi*), 167
- PyConfig.pathconfig_warnings (*C ögesi*), 167
- PyConfig.platlibdir (*C ögesi*), 165
- PyConfig.prefix (*C ögesi*), 167

PyConfig.program_name (*C ögesi*), 167
PyConfig.pycache_prefix (*C ögesi*), 167
PyConfig.pythonpath_env (*C ögesi*), 166
PyConfig.quiet (*C ögesi*), 167
PyConfig.run_command (*C ögesi*), 167
PyConfig.run_filename (*C ögesi*), 167
PyConfig.run_module (*C ögesi*), 167
PyConfig.show_ref_count (*C ögesi*), 167
PyConfig.site_import (*C ögesi*), 167
PyConfig.skip_source_first_line (*C ögesi*), 167
PyConfig.stdio_encoding (*C ögesi*), 167
PyConfig.stdio_errors (*C ögesi*), 167
PyConfig.tracemalloc (*C ögesi*), 167
PyConfig.use_environment (*C ögesi*), 167
PyConfig.use_hash_seed (*C ögesi*), 166
PyConfig.user_site_directory (*C ögesi*), 167
PyConfig.verbose (*C ögesi*), 167
PyConfig.warnoptions (*C ögesi*), 167
PyConfig.write_bytecode (*C ögesi*), 168
PyConfig.xoptions (*C ögesi*), 168
PyContext (*C tipi*), 133
PyContext_CheckExact (*C fonksiyonu*), 133
PyContext_Copy (*C fonksiyonu*), 134
PyContext_CopyCurrent (*C fonksiyonu*), 134
PyContext_Enter (*C fonksiyonu*), 134
PyContext_Exit (*C fonksiyonu*), 134
PyContext_New (*C fonksiyonu*), 134
PyContext_Type (*C değişkeni*), 133
PyContextToken (*C tipi*), 133
PyContextToken_CheckExact (*C fonksiyonu*), 133
PyContextToken_Type (*C değişkeni*), 133
PyContextVar (*C tipi*), 133
PyContextVar_CheckExact (*C fonksiyonu*), 133
PyContextVar_Get (*C fonksiyonu*), 134
PyContextVar_New (*C fonksiyonu*), 134
PyContextVar_Reset (*C fonksiyonu*), 134
PyContextVar_Set (*C fonksiyonu*), 134
PyContextVar_Type (*C değişkeni*), 133
PyCoro_CheckExact (*C fonksiyonu*), 133
PyCoro_New (*C fonksiyonu*), 133
PyCoro_Type (*C değişkeni*), 133
PyCoroObject (*C tipi*), 133
PyDate_Check (*C fonksiyonu*), 135
PyDate_CheckExact (*C fonksiyonu*), 135
PyDate_FromDate (*C fonksiyonu*), 135
PyDate_FromTimestamp (*C fonksiyonu*), 137
PyDateTime_Check (*C fonksiyonu*), 135
PyDateTime_CheckExact (*C fonksiyonu*), 135
PyDateTime_DATE_GET_FOLD (*C fonksiyonu*), 136
PyDateTime_DATE_GET_HOUR (*C fonksiyonu*), 136
PyDateTime_DATE_GET_MINUTE (*C fonksiyonu*), 136
PyDateTime_DATE_GET_SECOND (*C fonksiyonu*), 136
PyDateTime_DELTA_GET_DAYS (*C fonksiyonu*), 137
PyDateTime_DELTA_GET_MICROSECONDS (*C fonksiyonu*), 137
PyDateTime_DELTA_GET_SECONDS (*C fonksiyonu*), 137
PyDateTime_FromDateAndTime (*C fonksiyonu*), 135
PyDateTime_FromDateAndTimeAndFold (*C fonksiyonu*), 135
PyDateTime_FromTimestamp (*C fonksiyonu*), 137
PyDateTime_GET_DAY (*C fonksiyonu*), 136
PyDateTime_GET_MONTH (*C fonksiyonu*), 136
PyDateTime_GET_YEAR (*C fonksiyonu*), 136
PyDateTime_TIME_GET_FOLD (*C fonksiyonu*), 136
PyDateTime_TIME_GET_HOUR (*C fonksiyonu*), 136
PyDateTime_TIME_GET_MICROSECOND (*C fonksiyonu*), 136
PyDateTime_TIME_GET_MINUTE (*C fonksiyonu*), 136
PyDateTime_TIME_GET_SECOND (*C fonksiyonu*), 136
PyDateTime_TimeZone_UTC (*C değişkeni*), 134
PyDelta_Check (*C fonksiyonu*), 135
PyDelta_CheckExact (*C fonksiyonu*), 135
PyDelta_FromDSU (*C fonksiyonu*), 135
PyDescr_IsData (*C fonksiyonu*), 128
PyDescr_NewClassMethod (*C fonksiyonu*), 128
PyDescr_NewGetSet (*C fonksiyonu*), 127
PyDescr_NewMember (*C fonksiyonu*), 127
PyDescr_NewMethod (*C fonksiyonu*), 127
PyDescr_NewWrapper (*C fonksiyonu*), 128
PyDict_Check (*C fonksiyonu*), 113
PyDict_CheckExact (*C fonksiyonu*), 113
PyDict_Clear (*C fonksiyonu*), 113
PyDict_Contains (*C fonksiyonu*), 113
PyDict_Copy (*C fonksiyonu*), 113
PyDict_DelItem (*C fonksiyonu*), 113
PyDict_DelItemString (*C fonksiyonu*), 114
PyDict_GetItem (*C fonksiyonu*), 114
PyDict_GetItemString (*C fonksiyonu*), 114
PyDict_GetItemWithError (*C fonksiyonu*), 114
PyDict_Items (*C fonksiyonu*), 114
PyDict_Keys (*C fonksiyonu*), 114
PyDict_Merge (*C fonksiyonu*), 115
PyDict_MergeFromSeq2 (*C fonksiyonu*), 115
PyDict_New (*C fonksiyonu*), 113
PyDict_Next (*C fonksiyonu*), 114
PyDict_SetDefault (*C fonksiyonu*), 114
PyDict_SetItem (*C fonksiyonu*), 113
PyDict_SetItemString (*C fonksiyonu*), 113

- PyDict_Size (*C fonksiyonu*), 114
- PyDict_Type (*C değişkeni*), 113
- PyDict_Update (*C fonksiyonu*), 115
- PyDict_Values (*C fonksiyonu*), 114
- PyDictObject (*C tipi*), 113
- PyDictProxy_New (*C fonksiyonu*), 113
- PyDoc_STR (*C makrosu*), 5
- PyDoc_STRVAR (*C makrosu*), 5
- PyErr_BadArgument (*C fonksiyonu*), 24
- PyErr_BadInternalCall (*C fonksiyonu*), 26
- PyErr_CheckSignals (*C fonksiyonu*), 29
- PyErr_Clear (*C fonksiyonu*), 23
- PyErr_Clear(), 9, 11
- PyErr_ExceptionMatches (*C fonksiyonu*), 27
- PyErr_ExceptionMatches(), 11
- PyErr_Fetch (*C fonksiyonu*), 27
- PyErr_Format (*C fonksiyonu*), 24
- PyErr_FormatV (*C fonksiyonu*), 24
- PyErr_GetExcInfo (*C fonksiyonu*), 28
- PyErr_GivenExceptionMatches (*C fonksiyonu*), 27
- PyErr_NewException (*C fonksiyonu*), 29
- PyErr_NewExceptionWithDoc (*C fonksiyonu*), 29
- PyErr_NoMemory (*C fonksiyonu*), 24
- PyErr_NormalizeException (*C fonksiyonu*), 28
- PyErr_Occurred (*C fonksiyonu*), 27
- PyErr_Occurred(), 9
- PyErr_Print (*C fonksiyonu*), 24
- PyErr_PrintEx (*C fonksiyonu*), 23
- PyErr_ResourceWarning (*C fonksiyonu*), 27
- PyErr_Restore (*C fonksiyonu*), 27
- PyErr_SetExcFromWindowsErr (*C fonksiyonu*), 25
- PyErr_SetExcFromWindowsErrWithFilename (*C fonksiyonu*), 25
- PyErr_SetExcFromWindowsErrWithFilenameObject (*C fonksiyonu*), 25
- PyErr_SetExcFromWindowsErrWithFilenameObjectBytes (*C fonksiyonu*), 25
- PyErr_SetExcInfo (*C fonksiyonu*), 28
- PyErr_SetFromErrno (*C fonksiyonu*), 24
- PyErr_SetFromErrnoWithFilename (*C fonksiyonu*), 25
- PyErr_SetFromErrnoWithFilenameObject (*C fonksiyonu*), 24
- PyErr_SetFromErrnoWithFilenameObjects (*C fonksiyonu*), 25
- PyErr_SetFromWindowsErr (*C fonksiyonu*), 25
- PyErr_SetFromWindowsErrWithFilename (*C fonksiyonu*), 25
- PyErr_SetImportError (*C fonksiyonu*), 25
- PyErr_SetImportErrorSubclass (*C fonksiyonu*), 25
- PyErr_SetInterrupt (*C fonksiyonu*), 29
- PyErr_SetNone (*C fonksiyonu*), 24
- PyErr_SetObject (*C fonksiyonu*), 24
- PyErr_SetString (*C fonksiyonu*), 24
- PyErr_SetString(), 9
- PyErr_SyntaxLocation (*C fonksiyonu*), 26
- PyErr_SyntaxLocationEx (*C fonksiyonu*), 26
- PyErr_SyntaxLocationObject (*C fonksiyonu*), 26
- PyErr_WarnEx (*C fonksiyonu*), 26
- PyErr_WarnExplicit (*C fonksiyonu*), 26
- PyErr_WarnExplicitObject (*C fonksiyonu*), 26
- PyErr_WarnFormat (*C fonksiyonu*), 27
- PyErr_WriteUnraisable (*C fonksiyonu*), 24
- PyEval_AcquireLock (*C fonksiyonu*), 152
- PyEval_AcquireThread (*C fonksiyonu*), 151
- PyEval_AcquireThread(), 148
- PyEval_EvalCode (*C fonksiyonu*), 18
- PyEval_EvalCodeEx (*C fonksiyonu*), 18
- PyEval_EvalFrame (*C fonksiyonu*), 18
- PyEval_EvalFrameEx (*C fonksiyonu*), 18
- PyEval_GetBuiltins (*C fonksiyonu*), 53
- PyEval_GetFrame (*C fonksiyonu*), 53
- PyEval_GetFuncDesc (*C fonksiyonu*), 53
- PyEval_GetFuncName (*C fonksiyonu*), 53
- PyEval_GetGlobals (*C fonksiyonu*), 53
- PyEval_GetLocals (*C fonksiyonu*), 53
- PyEval_InitThreads (*C fonksiyonu*), 148
- PyEval_InitThreads(), 142
- PyEval_MergeCompilerFlags (*C fonksiyonu*), 19
- PyEval_ReleaseLock (*C fonksiyonu*), 152
- PyEval_ReleaseThread (*C fonksiyonu*), 152
- PyEval_ReleaseThread(), 148
- PyEval_RestoreThread (*C fonksiyonu*), 148
- PyEval_RestoreThread(), 146, 148
- PyEval_SaveThread (*C fonksiyonu*), 148
- PyEval_SaveThread(), 146, 148
- PyEval_SetProfile (*C fonksiyonu*), 155
- PyEval_SetTrace (*C fonksiyonu*), 155
- PyEval_ThreadsInitialized (*C fonksiyonu*), 148
- PyExc_ArithmeticError, 32
- PyExc_AssertionError, 32
- PyExc_AttributeError, 32
- PyExc_BaseException, 32
- PyExc_BlockingIOError, 32
- PyExc_BrokenPipeError, 32
- PyExc_BufferError, 32
- PyExc_BytesWarning, 33
- PyExc_ChildProcessError, 32
- PyExc_ConnectionAbortedError, 32
- PyExc_ConnectionError, 32
- PyExc_ConnectionRefusedError, 32
- PyExc_ConnectionResetError, 32
- PyExc_DeprecationWarning, 33
- PyExc_EnvironmentError, 33
- PyExc_EOFError, 32
- PyExc_Exception, 32
- PyExc_FileExistsError, 32
- PyExc_FileNotFoundError, 32
- PyExc_FloatingPointError, 32

PyExc_FutureWarning, 33
PyExc_GeneratorExit, 32
PyExc_ImportError, 32
PyExc_ImportWarning, 33
PyExc_IndentationError, 32
PyExc_IndexError, 32
PyExc_InterruptedError, 32
PyExc_IOError, 33
PyExc_IsADirectoryError, 32
PyExc_KeyboardInterrupt, 32
PyExc_KeyError, 32
PyExc_LookupError, 32
PyExc_MemoryError, 32
PyExc_ModuleNotFoundError, 32
PyExc_NameError, 32
PyExc_NotADirectoryError, 32
PyExc_NotImplementedError, 32
PyExc_OSError, 32
PyExc_OverflowError, 32
PyExc_PendingDeprecationWarning, 33
PyExc_PermissionError, 32
PyExc_ProcessLookupError, 32
PyExc_RecursionError, 32
PyExc_ReferenceError, 32
PyExc_ResourceWarning, 33
PyExc_RuntimeError, 32
PyExc_RuntimeWarning, 33
PyExc_StopAsyncIteration, 32
PyExc_StopIteration, 32
PyExc_SyntaxError, 32
PyExc_SyntaxWarning, 33
PyExc_SystemError, 32
PyExc_SystemExit, 32
PyExc_TabError, 32
PyExc_TimeoutError, 32
PyExc_TypeError, 32
PyExc_UnboundLocalError, 32
PyExc_UnicodeDecodeError, 32
PyExc_UnicodeEncodeError, 32
PyExc_UnicodeError, 32
PyExc_UnicodeTranslateError, 32
PyExc_UnicodeWarning, 33
PyExc_UserWarning, 33
PyExc_ValueError, 32
PyExc_Warning, 33
PyExc_WindowsError, 33
PyExc_ZeroDivisionError, 32
PyException_GetCause (C fonksiyonu), 30
PyException_GetContext (C fonksiyonu), 29
PyException_GetTraceback (C fonksiyonu), 29
PyException_SetCause (C fonksiyonu), 30
PyException_SetContext (C fonksiyonu), 29
PyException_SetTraceback (C fonksiyonu), 29
PyFile_FromFd (C fonksiyonu), 120
PyFile_GetLine (C fonksiyonu), 120
PyFile_SetOpenCodeHook (C fonksiyonu), 120
PyFile_WriteObject (C fonksiyonu), 121
PyFile_WriteString (C fonksiyonu), 121
PyFloat_AS_DOUBLE (C fonksiyonu), 86
PyFloat_AsDouble (C fonksiyonu), 86
PyFloat_Check (C fonksiyonu), 86
PyFloat_CheckExact (C fonksiyonu), 86
PyFloat_FromDouble (C fonksiyonu), 86
PyFloat_FromString (C fonksiyonu), 86
PyFloat_GetInfo (C fonksiyonu), 86
PyFloat_GetMax (C fonksiyonu), 87
PyFloat_GetMin (C fonksiyonu), 87
PyFloat_Type (C değişkeni), 86
PyFloatObject (C tipi), 86
PyFrame_GetBack (C fonksiyonu), 53
PyFrame_GetCode (C fonksiyonu), 53
PyFrame_GetLineNumber (C fonksiyonu), 53
PyFrameObject (C tipi), 18
PyFrozenSet_Check (C fonksiyonu), 116
PyFrozenSet_CheckExact (C fonksiyonu), 116
PyFrozenSet_New (C fonksiyonu), 116
PyFrozenSet_Type (C değişkeni), 116
PyFunction_Check (C fonksiyonu), 117
PyFunction_GetAnnotations (C fonksiyonu), 118
PyFunction_GetClosure (C fonksiyonu), 117
PyFunction_GetCode (C fonksiyonu), 117
PyFunction_GetDefaults (C fonksiyonu), 117
PyFunction_GetGlobals (C fonksiyonu), 117
PyFunction_GetModule (C fonksiyonu), 117
PyFunction_New (C fonksiyonu), 117
PyFunction_NewWithQualName (C fonksiyonu), 117
PyFunction_SetAnnotations (C fonksiyonu), 118
PyFunction_SetClosure (C fonksiyonu), 117
PyFunction_SetDefaults (C fonksiyonu), 117
PyFunction_Type (C değişkeni), 117
PyFunctionObject (C tipi), 117
PyGen_Check (C fonksiyonu), 132
PyGen_CheckExact (C fonksiyonu), 132
PyGen_New (C fonksiyonu), 132
PyGen_NewWithQualName (C fonksiyonu), 132
PyGen_Type (C değişkeni), 132
PyGenObject (C tipi), 132
PyGetSetDef (C tipi), 188
PyGILState_Check (C fonksiyonu), 149
PyGILState_Ensure (C fonksiyonu), 148
PyGILState_GetThisThreadState (C fonksiyonu), 149
PyGILState_Release (C fonksiyonu), 149
PyImport_AddModule (C fonksiyonu), 41
PyImport_AddModuleObject (C fonksiyonu), 40
PyImport_AppendInittab (C fonksiyonu), 42
PyImport_ExecCodeModule (C fonksiyonu), 41
PyImport_ExecCodeModuleEx (C fonksiyonu), 41
PyImport_ExecCodeModuleObject (C fonksiyonu), 41
PyImport_ExecCodeModuleWithPathnames (C fonksiyonu), 41

- PyImport_ExtendInittab (*C fonksiyonu*), 43
PyImport_FrozenModules (*C değişkeni*), 42
PyImport_GetImporter (*C fonksiyonu*), 42
PyImport_GetMagicNumber (*C fonksiyonu*), 41
PyImport_GetMagicTag (*C fonksiyonu*), 42
PyImport_GetModule (*C fonksiyonu*), 42
PyImport_GetModuleDict (*C fonksiyonu*), 42
PyImport_Import (*C fonksiyonu*), 40
PyImport_ImportFrozenModule (*C fonksiyonu*), 42
PyImport_ImportFrozenModuleObject (*C fonksiyonu*), 42
PyImport_ImportModule (*C fonksiyonu*), 40
PyImport_ImportModuleEx (*C fonksiyonu*), 40
PyImport_ImportModuleLevel (*C fonksiyonu*), 40
PyImport_ImportModuleLevelObject (*C fonksiyonu*), 40
PyImport_ImportModuleNoBlock (*C fonksiyonu*), 40
PyImport_ReloadModule (*C fonksiyonu*), 40
PyIndex_Check (*C fonksiyonu*), 68
PyInstanceMethod_Check (*C fonksiyonu*), 118
PyInstanceMethod_Function (*C fonksiyonu*), 118
PyInstanceMethod_GET_FUNCTION (*C fonksiyonu*), 118
PyInstanceMethod_New (*C fonksiyonu*), 118
PyInstanceMethod_Type (*C değişkeni*), 118
PyInterpreterState (*C tipi*), 148
PyInterpreterState_Clear (*C fonksiyonu*), 150
PyInterpreterState_Delete (*C fonksiyonu*), 150
PyInterpreterState_Get (*C fonksiyonu*), 150
PyInterpreterState_GetDict (*C fonksiyonu*), 151
PyInterpreterState_GetID (*C fonksiyonu*), 151
PyInterpreterState_Head (*C fonksiyonu*), 156
PyInterpreterState_Main (*C fonksiyonu*), 156
PyInterpreterState_New (*C fonksiyonu*), 150
PyInterpreterState_Next (*C fonksiyonu*), 156
PyInterpreterState_ThreadHead (*C fonksiyonu*), 156
PyIter_Check (*C fonksiyonu*), 71
PyIter_Next (*C fonksiyonu*), 71
PyList_Append (*C fonksiyonu*), 112
PyList_AsTuple (*C fonksiyonu*), 113
PyList_Check (*C fonksiyonu*), 112
PyList_CheckExact (*C fonksiyonu*), 112
PyList_GET_ITEM (*C fonksiyonu*), 112
PyList_GET_SIZE (*C fonksiyonu*), 112
PyList_GetItem (*C fonksiyonu*), 112
PyList_GetItem(), 8
PyList_GetSlice (*C fonksiyonu*), 112
PyList_Insert (*C fonksiyonu*), 112
PyList_New (*C fonksiyonu*), 112
PyList_Reverse (*C fonksiyonu*), 113
PyList_SET_ITEM (*C fonksiyonu*), 112
PyList_SetItem (*C fonksiyonu*), 112
PyList_SetItem(), 7
PyList_SetSlice (*C fonksiyonu*), 113
PyList_Size (*C fonksiyonu*), 112
PyList_Sort (*C fonksiyonu*), 113
PyList_Type (*C değişkeni*), 112
PyListObject (*C tipi*), 112
PyLong_AsDouble (*C fonksiyonu*), 85
PyLong_AsLong (*C fonksiyonu*), 84
PyLong_AsLongAndOverflow (*C fonksiyonu*), 84
PyLong_AsLongLong (*C fonksiyonu*), 84
PyLong_AsLongLongAndOverflow (*C fonksiyonu*), 84
PyLong_AsSize_t (*C fonksiyonu*), 85
PyLong_AsSsize_t (*C fonksiyonu*), 84
PyLong_AsUnsignedLong (*C fonksiyonu*), 85
PyLong_AsUnsignedLongLong (*C fonksiyonu*), 85
PyLong_AsUnsignedLongLongMask (*C fonksiyonu*), 85
PyLong_AsUnsignedLongMask (*C fonksiyonu*), 85
PyLong_AsVoidPtr (*C fonksiyonu*), 85
PyLong_Check (*C fonksiyonu*), 83
PyLong_CheckExact (*C fonksiyonu*), 83
PyLong_FromDouble (*C fonksiyonu*), 83
PyLong_FromLong (*C fonksiyonu*), 83
PyLong_FromLongLong (*C fonksiyonu*), 83
PyLong_FromSize_t (*C fonksiyonu*), 83
PyLong_FromSsize_t (*C fonksiyonu*), 83
PyLong_FromString (*C fonksiyonu*), 83
PyLong_FromUnicode (*C fonksiyonu*), 83
PyLong_FromUnicodeObject (*C fonksiyonu*), 84
PyLong_FromUnsignedLong (*C fonksiyonu*), 83
PyLong_FromUnsignedLongLong (*C fonksiyonu*), 83
PyLong_FromVoidPtr (*C fonksiyonu*), 84
PyLong_Type (*C değişkeni*), 83
PyLongObject (*C tipi*), 83
PyMapping_Check (*C fonksiyonu*), 70
PyMapping_DelItem (*C fonksiyonu*), 70
PyMapping_DelItemString (*C fonksiyonu*), 70
PyMapping_GetItemString (*C fonksiyonu*), 70
PyMapping_HasKey (*C fonksiyonu*), 70
PyMapping_HasKeyString (*C fonksiyonu*), 70
PyMapping_Items (*C fonksiyonu*), 70
PyMapping_Keys (*C fonksiyonu*), 70
PyMapping_Length (*C fonksiyonu*), 70
PyMapping_SetItemString (*C fonksiyonu*), 70
PyMapping_Size (*C fonksiyonu*), 70
PyMapping_Values (*C fonksiyonu*), 70
PyMappingMethods (*C tipi*), 214
PyMappingMethods.mp_ass_subscript (*C ögesi*), 214
PyMappingMethods.mp_length (*C ögesi*), 214

- `PyMappingMethods.mp_subscript` (*C ögesi*), 214
- `PyMarshal_ReadLastObjectFromFile` (*C fonksiyonu*), 43
- `PyMarshal_ReadLongFromFile` (*C fonksiyonu*), 43
- `PyMarshal_ReadObjectFromFile` (*C fonksiyonu*), 43
- `PyMarshal_ReadObjectFromString` (*C fonksiyonu*), 44
- `PyMarshal_ReadShortFromFile` (*C fonksiyonu*), 43
- `PyMarshal_WriteLongToFile` (*C fonksiyonu*), 43
- `PyMarshal_WriteObjectToFile` (*C fonksiyonu*), 43
- `PyMarshal_WriteObjectToString` (*C fonksiyonu*), 43
- `PyMem_Calloc` (*C fonksiyonu*), 177
- `PyMem_Del` (*C fonksiyonu*), 177
- `PYMEM_DOMAIN_MEM` (*C makrosu*), 179
- `PYMEM_DOMAIN_OBJ` (*C makrosu*), 180
- `PYMEM_DOMAIN_RAW` (*C makrosu*), 179
- `PyMem_Free` (*C fonksiyonu*), 177
- `PyMem_GetAllocator` (*C fonksiyonu*), 180
- `PyMem_Malloc` (*C fonksiyonu*), 177
- `PyMem_New` (*C fonksiyonu*), 177
- `PyMem_RawCalloc` (*C fonksiyonu*), 176
- `PyMem_RawFree` (*C fonksiyonu*), 176
- `PyMem_RawMalloc` (*C fonksiyonu*), 176
- `PyMem_RawRealloc` (*C fonksiyonu*), 176
- `PyMem_Realloc` (*C fonksiyonu*), 177
- `PyMem_Resize` (*C fonksiyonu*), 177
- `PyMem_SetAllocator` (*C fonksiyonu*), 180
- `PyMem_SetupDebugHooks` (*C fonksiyonu*), 180
- `PyMemAllocatorDomain` (*C tipi*), 179
- `PyMemAllocatorEx` (*C tipi*), 179
- `PyMember_GetOne` (*C fonksiyonu*), 188
- `PyMember_SetOne` (*C fonksiyonu*), 188
- `PyMemberDef` (*C tipi*), 187
- `PyMemoryView_Check` (*C fonksiyonu*), 129
- `PyMemoryView_FromBuffer` (*C fonksiyonu*), 129
- `PyMemoryView_FromMemory` (*C fonksiyonu*), 129
- `PyMemoryView_FromObject` (*C fonksiyonu*), 129
- `PyMemoryView_GET_BASE` (*C fonksiyonu*), 130
- `PyMemoryView_GET_BUFFER` (*C fonksiyonu*), 130
- `PyMemoryView_GetContiguous` (*C fonksiyonu*), 129
- `PyMethod_Check` (*C fonksiyonu*), 118
- `PyMethod_Function` (*C fonksiyonu*), 118
- `PyMethod_GET_FUNCTION` (*C fonksiyonu*), 118
- `PyMethod_GET_SELF` (*C fonksiyonu*), 119
- `PyMethod_New` (*C fonksiyonu*), 118
- `PyMethod_Self` (*C fonksiyonu*), 118
- `PyMethod_Type` (*C değişkeni*), 118
- `PyMethodDef` (*C tipi*), 186
- `PyModule_AddFunctions` (*C fonksiyonu*), 125
- `PyModule_AddIntConstant` (*C fonksiyonu*), 126
- `PyModule_AddIntMacro` (*C fonksiyonu*), 126
- `PyModule_AddObject` (*C fonksiyonu*), 126
- `PyModule_AddStringConstant` (*C fonksiyonu*), 126
- `PyModule_AddStringMacro` (*C fonksiyonu*), 126
- `PyModule_AddType` (*C fonksiyonu*), 126
- `PyModule_Check` (*C fonksiyonu*), 121
- `PyModule_CheckExact` (*C fonksiyonu*), 121
- `PyModule_Create` (*C fonksiyonu*), 123
- `PyModule_Create2` (*C fonksiyonu*), 123
- `PyModule_ExecDef` (*C fonksiyonu*), 125
- `PyModule_FromDefAndSpec` (*C fonksiyonu*), 125
- `PyModule_FromDefAndSpec2` (*C fonksiyonu*), 125
- `PyModule_GetDef` (*C fonksiyonu*), 122
- `PyModule_GetDict` (*C fonksiyonu*), 121
- `PyModule_GetFilename` (*C fonksiyonu*), 122
- `PyModule_GetFilenameObject` (*C fonksiyonu*), 122
- `PyModule_GetName` (*C fonksiyonu*), 121
- `PyModule_GetNameObject` (*C fonksiyonu*), 121
- `PyModule_GetState` (*C fonksiyonu*), 121
- `PyModule_New` (*C fonksiyonu*), 121
- `PyModule_NewObject` (*C fonksiyonu*), 121
- `PyModule_SetDocString` (*C fonksiyonu*), 125
- `PyModule_Type` (*C değişkeni*), 121
- `PyModuleDef` (*C tipi*), 122
- `PyModuleDef_Init` (*C fonksiyonu*), 124
- `PyModuleDef_Slot` (*C tipi*), 124
- `PyModuleDef_Slot.slot` (*C ögesi*), 124
- `PyModuleDef_Slot.value` (*C ögesi*), 124
- `PyModuleDef.m_base` (*C ögesi*), 122
- `PyModuleDef.m_clear` (*C ögesi*), 123
- `PyModuleDef.m_doc` (*C ögesi*), 122
- `PyModuleDef.m_free` (*C ögesi*), 123
- `PyModuleDef.m_methods` (*C ögesi*), 122
- `PyModuleDef.m_name` (*C ögesi*), 122
- `PyModuleDef.m_reload` (*C ögesi*), 123
- `PyModuleDef.m_size` (*C ögesi*), 122
- `PyModuleDef.m_slots` (*C ögesi*), 122
- `PyModuleDef.m_traverse` (*C ögesi*), 123
- `PyNumber_Absolute` (*C fonksiyonu*), 66
- `PyNumber_Add` (*C fonksiyonu*), 65
- `PyNumber_And` (*C fonksiyonu*), 66
- `PyNumber_AsSsize_t` (*C fonksiyonu*), 68
- `PyNumber_Check` (*C fonksiyonu*), 65
- `PyNumber_Divmod` (*C fonksiyonu*), 66
- `PyNumber_Float` (*C fonksiyonu*), 67
- `PyNumber_FloorDivide` (*C fonksiyonu*), 65
- `PyNumber_Index` (*C fonksiyonu*), 67
- `PyNumber_InPlaceAdd` (*C fonksiyonu*), 66
- `PyNumber_InPlaceAnd` (*C fonksiyonu*), 67
- `PyNumber_InPlaceFloorDivide` (*C fonksiyonu*), 67
- `PyNumber_InPlaceLshift` (*C fonksiyonu*), 67
- `PyNumber_InPlaceMatrixMultiply` (*C fonksiyonu*), 67
- `PyNumber_InPlaceMultiply` (*C fonksiyonu*), 67

- PyNumber_InPlaceOr (*C fonksiyonu*), 67
- PyNumber_InPlacePower (*C fonksiyonu*), 67
- PyNumber_InPlaceRemainder (*C fonksiyonu*), 67
- PyNumber_InPlaceRshift (*C fonksiyonu*), 67
- PyNumber_InPlaceSubtract (*C fonksiyonu*), 66
- PyNumber_InPlaceTrueDivide (*C fonksiyonu*), 67
- PyNumber_InPlaceXor (*C fonksiyonu*), 67
- PyNumber_Invert (*C fonksiyonu*), 66
- PyNumber_Long (*C fonksiyonu*), 67
- PyNumber_Lshift (*C fonksiyonu*), 66
- PyNumber_MatrixMultiply (*C fonksiyonu*), 65
- PyNumber_Multiply (*C fonksiyonu*), 65
- PyNumber_Negative (*C fonksiyonu*), 66
- PyNumber_Or (*C fonksiyonu*), 66
- PyNumber_Positive (*C fonksiyonu*), 66
- PyNumber_Power (*C fonksiyonu*), 66
- PyNumber_Remainder (*C fonksiyonu*), 66
- PyNumber_Rshift (*C fonksiyonu*), 66
- PyNumber_Subtract (*C fonksiyonu*), 65
- PyNumber_ToBase (*C fonksiyonu*), 67
- PyNumber_TrueDivide (*C fonksiyonu*), 66
- PyNumber_Xor (*C fonksiyonu*), 66
- PyNumberMethods (*C tipi*), 212
- PyNumberMethods.nb_absolute (*C ögesi*), 213
- PyNumberMethods.nb_add (*C ögesi*), 213
- PyNumberMethods.nb_and (*C ögesi*), 213
- PyNumberMethods.nb_bool (*C ögesi*), 213
- PyNumberMethods.nb_divmod (*C ögesi*), 213
- PyNumberMethods.nb_float (*C ögesi*), 213
- PyNumberMethods.nb_floor_divide (*C ögesi*), 214
- PyNumberMethods.nb_index (*C ögesi*), 214
- PyNumberMethods.nb_inplace_add (*C ögesi*), 213
- PyNumberMethods.nb_inplace_and (*C ögesi*), 214
- PyNumberMethods.nb_inplace_floor_divide (*C ögesi*), 214
- PyNumberMethods.nb_inplace_lshift (*C ögesi*), 214
- PyNumberMethods.nb_inplace_matrix_multiply (*C ögesi*), 214
- PyNumberMethods.nb_inplace_multiply (*C ögesi*), 213
- PyNumberMethods.nb_inplace_or (*C ögesi*), 214
- PyNumberMethods.nb_inplace_power (*C ögesi*), 213
- PyNumberMethods.nb_inplace_remainder (*C ögesi*), 213
- PyNumberMethods.nb_inplace_rshift (*C ögesi*), 214
- PyNumberMethods.nb_inplace_subtract (*C ögesi*), 213
- PyNumberMethods.nb_inplace_true_divide (*C ögesi*), 214
- PyNumberMethods.nb_inplace_xor (*C ögesi*), 214
- PyNumberMethods.nb_int (*C ögesi*), 213
- PyNumberMethods.nb_invert (*C ögesi*), 213
- PyNumberMethods.nb_lshift (*C ögesi*), 213
- PyNumberMethods.nb_matrix_multiply (*C ögesi*), 214
- PyNumberMethods.nb_multiply (*C ögesi*), 213
- PyNumberMethods.nb_negative (*C ögesi*), 213
- PyNumberMethods.nb_or (*C ögesi*), 213
- PyNumberMethods.nb_positive (*C ögesi*), 213
- PyNumberMethods.nb_power (*C ögesi*), 213
- PyNumberMethods.nb_remainder (*C ögesi*), 213
- PyNumberMethods.nb_reserved (*C ögesi*), 213
- PyNumberMethods.nb_rshift (*C ögesi*), 213
- PyNumberMethods.nb_subtract (*C ögesi*), 213
- PyNumberMethods.nb_true_divide (*C ögesi*), 214
- PyNumberMethods.nb_xor (*C ögesi*), 213
- PyObject (*C tipi*), 184
- PyObject_AsCharBuffer (*C fonksiyonu*), 78
- PyObject_ASCII (*C fonksiyonu*), 59
- PyObject_AsFileDescriptor (*C fonksiyonu*), 120
- PyObject_AsReadBuffer (*C fonksiyonu*), 78
- PyObject_AsWriteBuffer (*C fonksiyonu*), 78
- PyObject_Bytes (*C fonksiyonu*), 59
- PyObject_Call (*C fonksiyonu*), 63
- PyObject_CallFunction (*C fonksiyonu*), 63
- PyObject_CallFunctionObjArgs (*C fonksiyonu*), 64
- PyObject_CallMethod (*C fonksiyonu*), 64
- PyObject_CallMethodNoArgs (*C fonksiyonu*), 64
- PyObject_CallMethodObjArgs (*C fonksiyonu*), 64
- PyObject_CallMethodOneArg (*C fonksiyonu*), 64
- PyObject_CallNoArgs (*C fonksiyonu*), 63
- PyObject_CallObject (*C fonksiyonu*), 63
- PyObject_Calloc (*C fonksiyonu*), 178
- PyObject_CallOneArg (*C fonksiyonu*), 63
- PyObject_CheckBuffer (*C fonksiyonu*), 77
- PyObject_CheckReadBuffer (*C fonksiyonu*), 78
- PyObject_Del (*C fonksiyonu*), 183
- PyObject_DelAttr (*C fonksiyonu*), 58
- PyObject_DelAttrString (*C fonksiyonu*), 58
- PyObject_DelItem (*C fonksiyonu*), 60
- PyObject_Dir (*C fonksiyonu*), 60
- PyObject_Free (*C fonksiyonu*), 178
- PyObject_GC_Del (*C fonksiyonu*), 221
- PyObject_GC_IsFinalized (*C fonksiyonu*), 221
- PyObject_GC_IsTracked (*C fonksiyonu*), 221
- PyObject_GC_New (*C fonksiyonu*), 221
- PyObject_GC_NewVar (*C fonksiyonu*), 221
- PyObject_GC_Resize (*C fonksiyonu*), 221
- PyObject_GC_Track (*C fonksiyonu*), 221

- PyObject_GC_UnTrack (*C fonksiyonu*), 221
- PyObject_GenericGetAttr (*C fonksiyonu*), 58
- PyObject_GenericGetDict (*C fonksiyonu*), 58
- PyObject_GenericSetAttr (*C fonksiyonu*), 58
- PyObject_GenericSetDict (*C fonksiyonu*), 58
- PyObject_GetArenaAllocator (*C fonksiyonu*), 181
- PyObject_GetAttr (*C fonksiyonu*), 57
- PyObject_GetAttrString (*C fonksiyonu*), 58
- PyObject_GetBuffer (*C fonksiyonu*), 77
- PyObject_GetItem (*C fonksiyonu*), 60
- PyObject_GetIter (*C fonksiyonu*), 60
- PyObject_HasAttr (*C fonksiyonu*), 57
- PyObject_HasAttrString (*C fonksiyonu*), 57
- PyObject_Hash (*C fonksiyonu*), 59
- PyObject_HashNotImplemented (*C fonksiyonu*), 60
- PyObject_HEAD (*C makrosu*), 184
- PyObject_HEAD_INIT (*C makrosu*), 185
- PyObject_Init (*C fonksiyonu*), 183
- PyObject_InitVar (*C fonksiyonu*), 183
- PyObject_IS_GC (*C fonksiyonu*), 221
- PyObject_IsInstance (*C fonksiyonu*), 59
- PyObject_IsSubclass (*C fonksiyonu*), 59
- PyObject_IsTrue (*C fonksiyonu*), 60
- PyObject_Length (*C fonksiyonu*), 60
- PyObject_LengthHint (*C fonksiyonu*), 60
- PyObject_Malloc (*C fonksiyonu*), 178
- PyObject_New (*C fonksiyonu*), 183
- PyObject_NewVar (*C fonksiyonu*), 183
- PyObject_Not (*C fonksiyonu*), 60
- PyObject._ob_next (*C ögesi*), 195
- PyObject._ob_prev (*C ögesi*), 195
- PyObject_Print (*C fonksiyonu*), 57
- PyObject_Realloc (*C fonksiyonu*), 178
- PyObject_Repr (*C fonksiyonu*), 59
- PyObject_RichCompare (*C fonksiyonu*), 58
- PyObject_RichCompareBool (*C fonksiyonu*), 58
- PyObject_SetArenaAllocator (*C fonksiyonu*), 181
- PyObject_SetAttr (*C fonksiyonu*), 58
- PyObject_SetAttrString (*C fonksiyonu*), 58
- PyObject_SetItem (*C fonksiyonu*), 60
- PyObject_Size (*C fonksiyonu*), 60
- PyObject_Str (*C fonksiyonu*), 59
- PyObject_Type (*C fonksiyonu*), 60
- PyObject_TypeCheck (*C fonksiyonu*), 60
- PyObject_VAR_HEAD (*C makrosu*), 184
- PyObject_Vectorcall (*C fonksiyonu*), 64
- PyObject_VectorcallDict (*C fonksiyonu*), 64
- PyObject_VectorcallMethod (*C fonksiyonu*), 65
- PyObjectArenaAllocator (*C tipi*), 181
- PyObject.ob_refcnt (*C ögesi*), 195
- PyObject.ob_type (*C ögesi*), 195
- PyOS_AfterFork (*C fonksiyonu*), 36
- PyOS_AfterFork_Child (*C fonksiyonu*), 36
- PyOS_AfterFork_Parent (*C fonksiyonu*), 35
- PyOS_BeforeFork (*C fonksiyonu*), 35
- PyOS_CheckStack (*C fonksiyonu*), 36
- PyOS_double_to_string (*C fonksiyonu*), 52
- PyOS_FSPath (*C fonksiyonu*), 35
- PyOS_getsig (*C fonksiyonu*), 36
- PyOS_InputHook (*C değişkeni*), 16
- PyOS_ReadlineFunctionPointer (*C değişkeni*), 16
- PyOS_setsig (*C fonksiyonu*), 36
- PyOS_snprintf (*C fonksiyonu*), 51
- PyOS_stricmp (*C fonksiyonu*), 52
- PyOS_string_to_double (*C fonksiyonu*), 51
- PyOS_strnicmp (*C fonksiyonu*), 52
- PyOS_vsnprintf (*C fonksiyonu*), 51
- PyParser_SimpleParseFile (*C fonksiyonu*), 17
- PyParser_SimpleParseFileFlags (*C fonksiyonu*), 17
- PyParser_SimpleParseString (*C fonksiyonu*), 17
- PyParser_SimpleParseStringFlags (*C fonksiyonu*), 17
- PyParser_SimpleParseStringFlagsFilename (*C fonksiyonu*), 17
- PyPreConfig (*C tipi*), 162
- PyPreConfig_InitIsolatedConfig (*C fonksiyonu*), 162
- PyPreConfig_InitPythonConfig (*C fonksiyonu*), 162
- PyPreConfig allocator (*C ögesi*), 162
- PyPreConfig.coerce_c_locale (*C ögesi*), 162
- PyPreConfig.coerce_c_locale_warn (*C ögesi*), 162
- PyPreConfig.configure_locale (*C ögesi*), 162
- PyPreConfig.dev_mode (*C ögesi*), 162
- PyPreConfig.isolated (*C ögesi*), 162
- PyPreConfig.legacy_windows_fs_encoding (*C ögesi*), 163
- PyPreConfig.parse_argv (*C ögesi*), 163
- PyPreConfig.use_environment (*C ögesi*), 163
- PyPreConfig.utf8_mode (*C ögesi*), 163
- PyProperty_Type (*C değişkeni*), 127
- PyRun_AnyFile (*C fonksiyonu*), 15
- PyRun_AnyFileEx (*C fonksiyonu*), 15
- PyRun_AnyFileExFlags (*C fonksiyonu*), 15
- PyRun_AnyFileFlags (*C fonksiyonu*), 15
- PyRun_File (*C fonksiyonu*), 17
- PyRun_FileEx (*C fonksiyonu*), 17
- PyRun_FileExFlags (*C fonksiyonu*), 18
- PyRun_FileFlags (*C fonksiyonu*), 17
- PyRun_InteractiveLoop (*C fonksiyonu*), 16
- PyRun_InteractiveLoopFlags (*C fonksiyonu*), 16
- PyRun_InteractiveOne (*C fonksiyonu*), 16
- PyRun_InteractiveOneFlags (*C fonksiyonu*), 16
- PyRun_SimpleFile (*C fonksiyonu*), 16
- PyRun_SimpleFileEx (*C fonksiyonu*), 16

- PyRun_SimpleFileExFlags (*C fonksiyonu*), 16
- PyRun_SimpleString (*C fonksiyonu*), 16
- PyRun_SimpleStringFlags (*C fonksiyonu*), 16
- PyRun_String (*C fonksiyonu*), 17
- PyRun_StringFlags (*C fonksiyonu*), 17
- PySeqIter_Check (*C fonksiyonu*), 127
- PySeqIter_New (*C fonksiyonu*), 127
- PySeqIter_Type (*C değişkeni*), 127
- PySequence_Check (*C fonksiyonu*), 68
- PySequence_Concat (*C fonksiyonu*), 68
- PySequence_Contains (*C fonksiyonu*), 69
- PySequence_Count (*C fonksiyonu*), 69
- PySequence_DelItem (*C fonksiyonu*), 68
- PySequence_DelSlice (*C fonksiyonu*), 69
- PySequence_Fast (*C fonksiyonu*), 69
- PySequence_Fast_GET_ITEM (*C fonksiyonu*), 69
- PySequence_Fast_GET_SIZE (*C fonksiyonu*), 69
- PySequence_Fast_ITEMS (*C fonksiyonu*), 69
- PySequence_GetItem (*C fonksiyonu*), 68
- PySequence_GetItem(), 8
- PySequence_GetSlice (*C fonksiyonu*), 68
- PySequence_Index (*C fonksiyonu*), 69
- PySequence_InPlaceConcat (*C fonksiyonu*), 68
- PySequence_InPlaceRepeat (*C fonksiyonu*), 68
- PySequence_ITEM (*C fonksiyonu*), 69
- PySequence_Length (*C fonksiyonu*), 68
- PySequence_List (*C fonksiyonu*), 69
- PySequence_Repeat (*C fonksiyonu*), 68
- PySequence_SetItem (*C fonksiyonu*), 68
- PySequence_SetSlice (*C fonksiyonu*), 68
- PySequence_Size (*C fonksiyonu*), 68
- PySequence_Tuple (*C fonksiyonu*), 69
- PySequenceMethods (*C tipi*), 214
- PySequenceMethods.sq_ass_item (*C ögesi*), 215
- PySequenceMethods.sq_concat (*C ögesi*), 214
- PySequenceMethods.sq_contains (*C ögesi*), 215
- PySequenceMethods.sq_inplace_concat (*C ögesi*), 215
- PySequenceMethods.sq_inplace_repeat (*C ögesi*), 215
- PySequenceMethods.sq_item (*C ögesi*), 214
- PySequenceMethods.sq_length (*C ögesi*), 214
- PySequenceMethods.sq_repeat (*C ögesi*), 214
- PySet_Add (*C fonksiyonu*), 116
- PySet_Check (*C fonksiyonu*), 116
- PySet_Clear (*C fonksiyonu*), 117
- PySet_Contains (*C fonksiyonu*), 116
- PySet_Discard (*C fonksiyonu*), 116
- PySet_GET_SIZE (*C fonksiyonu*), 116
- PySet_New (*C fonksiyonu*), 116
- PySet_Pop (*C fonksiyonu*), 116
- PySet_Size (*C fonksiyonu*), 116
- PySet_Type (*C değişkeni*), 115
- PySetObject (*C tipi*), 115
- PySignal_SetWakeupFd (*C fonksiyonu*), 29
- PySlice_AdjustIndices (*C fonksiyonu*), 129
- PySlice_Check (*C fonksiyonu*), 128
- PySlice_GetIndices (*C fonksiyonu*), 128
- PySlice_GetIndicesEx (*C fonksiyonu*), 128
- PySlice_New (*C fonksiyonu*), 128
- PySlice_Type (*C değişkeni*), 128
- PySlice_Unpack (*C fonksiyonu*), 129
- PyState_AddModule (*C fonksiyonu*), 126
- PyState_FindModule (*C fonksiyonu*), 126
- PyState_RemoveModule (*C fonksiyonu*), 127
- PyStatus (*C tipi*), 161
- PyStatus_Error (*C fonksiyonu*), 161
- PyStatus_Exception (*C fonksiyonu*), 161
- PyStatus_Exit (*C fonksiyonu*), 161
- PyStatus_IsError (*C fonksiyonu*), 161
- PyStatus_IsExit (*C fonksiyonu*), 161
- PyStatus_NoMemory (*C fonksiyonu*), 161
- PyStatus_Ok (*C fonksiyonu*), 161
- PyStatus.err_msg (*C ögesi*), 161
- PyStatus.exitcode (*C ögesi*), 161
- PyStatus.func (*C ögesi*), 161
- PyStructSequence_Desc (*C tipi*), 111
- PyStructSequence_Field (*C tipi*), 111
- PyStructSequence_GET_ITEM (*C fonksiyonu*), 111
- PyStructSequence_GetItem (*C fonksiyonu*), 111
- PyStructSequence_InitType (*C fonksiyonu*), 110
- PyStructSequence_InitType2 (*C fonksiyonu*), 110
- PyStructSequence_New (*C fonksiyonu*), 111
- PyStructSequence_NewType (*C fonksiyonu*), 110
- PyStructSequence_SET_ITEM (*C fonksiyonu*), 111
- PyStructSequence_SetItem (*C fonksiyonu*), 111
- PyStructSequence_UnnamedField (*C değişkeni*), 111
- PySys_AddAuditHook (*C fonksiyonu*), 39
- PySys_AddWarnOption (*C fonksiyonu*), 38
- PySys_AddWarnOptionUnicode (*C fonksiyonu*), 38
- PySys_AddXOption (*C fonksiyonu*), 38
- PySys_Audit (*C fonksiyonu*), 38
- PySys_FormatStderr (*C fonksiyonu*), 38
- PySys_FormatStdout (*C fonksiyonu*), 38
- PySys_GetObject (*C fonksiyonu*), 37
- PySys_GetXOptions (*C fonksiyonu*), 38
- PySys_ResetWarnOptions (*C fonksiyonu*), 37
- PySys_SetArgv (*C fonksiyonu*), 145
- PySys_SetArgv(), 142
- PySys_SetArgvEx (*C fonksiyonu*), 145
- PySys_SetArgvEx(), 11, 142
- PySys_SetObject (*C fonksiyonu*), 37
- PySys_SetPath (*C fonksiyonu*), 38
- PySys_WriteStderr (*C fonksiyonu*), 38
- PySys_WriteStdout (*C fonksiyonu*), 38

- Python 3000, [234](#)
- PYTHON*, [140](#)
- PYTHONCOERCECLOCALE, [170](#)
- PYTHONDEBUG, [140](#)
- PYTHONDONTWRITEBYTECODE, [140](#)
- PYTHONDUMPPREFS, [195](#)
- PYTHONHASHSEED, [140](#)
- PYTHONHOME, [11](#), [140](#), [145](#), [166](#)
- Python'ı İyileştirme Önerileri
 - PEP 1, [234](#)
 - PEP 7, [3](#), [5](#)
 - PEP 238, [19](#), [229](#)
 - PEP 278, [236](#)
 - PEP 302, [229](#), [232](#)
 - PEP 343, [227](#)
 - PEP 353, [9](#)
 - PEP 362, [226](#), [233](#)
 - PEP 383, [99](#), [100](#)
 - PEP 384, [13](#)
 - PEP 393, [91](#), [98](#)
 - PEP 411, [234](#)
 - PEP 420, [229](#), [233](#), [234](#)
 - PEP 432, [172](#), [173](#)
 - PEP 442, [211](#)
 - PEP 443, [229](#)
 - PEP 451, [124](#), [229](#)
 - PEP 483, [230](#)
 - PEP 484, [225](#), [229](#), [230](#), [236](#)
 - PEP 489, [125](#)
 - PEP 492, [226](#), [227](#)
 - PEP 498, [228](#)
 - PEP 519, [234](#)
 - PEP 523, [151](#)
 - PEP 525, [226](#)
 - PEP 526, [225](#), [236](#)
 - PEP 528, [141](#)
 - PEP 529, [100](#), [141](#)
 - PEP 538, [170](#)
 - PEP 539, [156](#)
 - PEP 540, [170](#)
 - PEP 552, [165](#)
 - PEP 578, [39](#)
 - PEP 585, [230](#)
 - PEP 587, [160](#)
 - PEP 590, [61](#)
 - PEP 617, [168](#)
 - PEP 623, [91](#)
 - PEP 3116, [236](#)
 - PEP 3119, [59](#)
 - PEP 3121, [122](#)
 - PEP 3147, [42](#)
 - PEP 3151, [33](#)
 - PEP 3155, [234](#)
- PYTHONINSPECT, [141](#)
- PYTHONIOENCODING, [143](#)
- Pythonic, [234](#)
- PYTHONLEGACYWINDOWSFSENCODING, [141](#)
- PYTHONLEGACYWINDOWSSTDIO, [141](#)
- PYTHONMALLOC, [176](#), [179](#), [180](#)
- PYTHONMALLOCSTATS, [176](#)
- PYTHONNOUSERSITE, [141](#)
- PYTHONOLDPARSER, [168](#)
- PYTHONOPTIMIZE, [141](#)
- PYTHONPATH, [11](#), [140](#), [166](#)
- Python'un Zen'i, [237](#)
- PYTHONUNBUFFERED, [141](#)
- PYTHONUTF8, [170](#)
- PYTHONVERBOSE, [141](#)
- PyThread_create_key (*C fonksiyonu*), [157](#)
- PyThread_delete_key (*C fonksiyonu*), [157](#)
- PyThread_delete_key_value (*C fonksiyonu*), [158](#)
- PyThread_get_key_value (*C fonksiyonu*), [157](#)
- PyThread_ReInitTLS (*C fonksiyonu*), [158](#)
- PyThread_set_key_value (*C fonksiyonu*), [157](#)
- PyThread_tss_alloc (*C fonksiyonu*), [157](#)
- PyThread_tss_create (*C fonksiyonu*), [157](#)
- PyThread_tss_delete (*C fonksiyonu*), [157](#)
- PyThread_tss_free (*C fonksiyonu*), [157](#)
- PyThread_tss_get (*C fonksiyonu*), [157](#)
- PyThread_tss_is_created (*C fonksiyonu*), [157](#)
- PyThread_tss_set (*C fonksiyonu*), [157](#)
- PyThreadState, [146](#)
- PyThreadState (*C tipi*), [148](#)
- PyThreadState_Clear (*C fonksiyonu*), [150](#)
- PyThreadState_Delete (*C fonksiyonu*), [150](#)
- PyThreadState_DeleteCurrent (*C fonksiyonu*), [150](#)
- PyThreadState_Get (*C fonksiyonu*), [148](#)
- PyThreadState_GetDict (*C fonksiyonu*), [151](#)
- PyThreadState_GetFrame (*C fonksiyonu*), [150](#)
- PyThreadState_GetID (*C fonksiyonu*), [150](#)
- PyThreadState_GetInterpreter (*C fonksiyonu*), [150](#)
- PyThreadState_New (*C fonksiyonu*), [150](#)
- PyThreadState_Next (*C fonksiyonu*), [156](#)
- PyThreadState_SetAsyncExc (*C fonksiyonu*), [151](#)
- PyThreadState_Swap (*C fonksiyonu*), [148](#)
- PyTime_Check (*C fonksiyonu*), [135](#)
- PyTime_CheckExact (*C fonksiyonu*), [135](#)
- PyTime_FromTime (*C fonksiyonu*), [135](#)
- PyTime_FromTimeAndFold (*C fonksiyonu*), [135](#)
- PyTimeZone_FromOffset (*C fonksiyonu*), [136](#)
- PyTimeZone_FromOffsetAndName (*C fonksiyonu*), [136](#)
- PyTrace_C_CALL (*C değişkeni*), [155](#)
- PyTrace_C_EXCEPTION (*C değişkeni*), [155](#)
- PyTrace_C_RETURN (*C değişkeni*), [155](#)
- PyTrace_CALL (*C değişkeni*), [155](#)
- PyTrace_EXCEPTION (*C değişkeni*), [155](#)
- PyTrace_LINE (*C değişkeni*), [155](#)
- PyTrace_OPCODE (*C değişkeni*), [155](#)
- PyTrace_RETURN (*C değişkeni*), [155](#)
- PyTraceMalloc_Track (*C fonksiyonu*), [181](#)
- PyTraceMalloc_Untrack (*C fonksiyonu*), [181](#)

- `PyTuple_Check` (*C fonksiyonu*), 109
- `PyTuple_CheckExact` (*C fonksiyonu*), 109
- `PyTuple_GET_ITEM` (*C fonksiyonu*), 110
- `PyTuple_GET_SIZE` (*C fonksiyonu*), 110
- `PyTuple_GetItem` (*C fonksiyonu*), 110
- `PyTuple_GetSlice` (*C fonksiyonu*), 110
- `PyTuple_New` (*C fonksiyonu*), 109
- `PyTuple_Pack` (*C fonksiyonu*), 109
- `PyTuple_SET_ITEM` (*C fonksiyonu*), 110
- `PyTuple_SetItem` (*C fonksiyonu*), 110
- `PyTuple_SetItem()`, 7
- `PyTuple_Size` (*C fonksiyonu*), 110
- `PyTuple_Type` (*C değişkeni*), 109
- `PyTupleObject` (*C tipi*), 109
- `PyType_Check` (*C fonksiyonu*), 79
- `PyType_CheckExact` (*C fonksiyonu*), 79
- `PyType_ClearCache` (*C fonksiyonu*), 79
- `PyType_FromModuleAndSpec` (*C fonksiyonu*), 81
- `PyType_FromSpec` (*C fonksiyonu*), 81
- `PyType_FromSpecWithBases` (*C fonksiyonu*), 81
- `PyType_GenericAlloc` (*C fonksiyonu*), 80
- `PyType_GenericNew` (*C fonksiyonu*), 80
- `PyType_GetFlags` (*C fonksiyonu*), 79
- `PyType_GetModule` (*C fonksiyonu*), 80
- `PyType_GetModuleState` (*C fonksiyonu*), 80
- `PyType_GetSlot` (*C fonksiyonu*), 80
- `PyType_HasFeature` (*C fonksiyonu*), 80
- `PyType_IS_GC` (*C fonksiyonu*), 80
- `PyType_IsSubtype` (*C fonksiyonu*), 80
- `PyType_Modified` (*C fonksiyonu*), 80
- `PyType_Ready` (*C fonksiyonu*), 80
- `PyType_Slot` (*C tipi*), 81
- `PyType_Slot.PyType_Slot.pfunc` (*C ögesi*), 82
- `PyType_Slot.PyType_Slot.slot` (*C ögesi*), 81
- `PyType_Spec` (*C tipi*), 81
- `PyType_Spec.PyType_Spec.basicsize` (*C ögesi*), 81
- `PyType_Spec.PyType_Spec.flags` (*C ögesi*), 81
- `PyType_Spec.PyType_Spec.itemsize` (*C ögesi*), 81
- `PyType_Spec.PyType_Spec.name` (*C ögesi*), 81
- `PyType_Spec.PyType_Spec.slots` (*C ögesi*), 81
- `PyType_Type` (*C değişkeni*), 79
- `PyTypeObject` (*C tipi*), 79
- `PyTypeObject.tp_alloc` (*C ögesi*), 209
- `PyTypeObject.tp_as_async` (*C ögesi*), 198
- `PyTypeObject.tp_as_buffer` (*C ögesi*), 201
- `PyTypeObject.tp_as_mapping` (*C ögesi*), 199
- `PyTypeObject.tp_as_number` (*C ögesi*), 199
- `PyTypeObject.tp_as_sequence` (*C ögesi*), 199
- `PyTypeObject.tp_base` (*C ögesi*), 207
- `PyTypeObject.tp_bases` (*C ögesi*), 210
- `PyTypeObject.tp_basicsize` (*C ögesi*), 196
- `PyTypeObject.tp_cache` (*C ögesi*), 210
- `PyTypeObject.tp_call` (*C ögesi*), 200
- `PyTypeObject.tp_clear` (*C ögesi*), 204
- `PyTypeObject.tp_dealloc` (*C ögesi*), 197
- `PyTypeObject.tp_del` (*C ögesi*), 210
- `PyTypeObject.tp_descr_get` (*C ögesi*), 207
- `PyTypeObject.tp_descr_set` (*C ögesi*), 207
- `PyTypeObject.tp_dict` (*C ögesi*), 207
- `PyTypeObject.tp_dictoffset` (*C ögesi*), 208
- `PyTypeObject.tp_doc` (*C ögesi*), 203
- `PyTypeObject.tp_finalize` (*C ögesi*), 211
- `PyTypeObject.tp_flags` (*C ögesi*), 201
- `PyTypeObject.tp_free` (*C ögesi*), 209
- `PyTypeObject.tp_getattr` (*C ögesi*), 198
- `PyTypeObject.tp_getattro` (*C ögesi*), 200
- `PyTypeObject.tp_getset` (*C ögesi*), 206
- `PyTypeObject.tp_hash` (*C ögesi*), 199
- `PyTypeObject.tp_init` (*C ögesi*), 208
- `PyTypeObject.tp_is_gc` (*C ögesi*), 210
- `PyTypeObject.tp_itemsize` (*C ögesi*), 196
- `PyTypeObject.tp_iter` (*C ögesi*), 206
- `PyTypeObject.tp_iternext` (*C ögesi*), 206
- `PyTypeObject.tp_members` (*C ögesi*), 206
- `PyTypeObject.tp_methods` (*C ögesi*), 206
- `PyTypeObject.tp_mro` (*C ögesi*), 210
- `PyTypeObject.tp_name` (*C ögesi*), 196
- `PyTypeObject.tp_new` (*C ögesi*), 209
- `PyTypeObject.tp_repr` (*C ögesi*), 199
- `PyTypeObject.tp_richcompare` (*C ögesi*), 205
- `PyTypeObject.tp_setattr` (*C ögesi*), 198
- `PyTypeObject.tp_setattro` (*C ögesi*), 200
- `PyTypeObject.tp_str` (*C ögesi*), 200
- `PyTypeObject.tp_subclasses` (*C ögesi*), 210
- `PyTypeObject.tp_traverse` (*C ögesi*), 203
- `PyTypeObject.tp_vectorcall` (*C ögesi*), 211
- `PyTypeObject.tp_vectorcall_offset` (*C ögesi*), 198
- `PyTypeObject.tp_version_tag` (*C ögesi*), 211
- `PyTypeObject.tp_weaklist` (*C ögesi*), 210
- `PyTypeObject.tp_weaklistoffset` (*C ögesi*), 205
- `PyTZInfo_Check` (*C fonksiyonu*), 135
- `PyTZInfo_CheckExact` (*C fonksiyonu*), 135
- `PyUnicode_1BYTE_DATA` (*C fonksiyonu*), 92
- `PyUnicode_1BYTE_KIND` (*C makrosu*), 92
- `PyUnicode_2BYTE_DATA` (*C fonksiyonu*), 92
- `PyUnicode_2BYTE_KIND` (*C makrosu*), 92
- `PyUnicode_4BYTE_DATA` (*C fonksiyonu*), 92
- `PyUnicode_4BYTE_KIND` (*C makrosu*), 92
- `PyUnicode_AS_DATA` (*C fonksiyonu*), 93
- `PyUnicode_AS_UNICODE` (*C fonksiyonu*), 93
- `PyUnicode_AsASCIIString` (*C fonksiyonu*), 106
- `PyUnicode_AsCharmapString` (*C fonksiyonu*), 107
- `PyUnicode_AsEncodedString` (*C fonksiyonu*), 102
- `PyUnicode_AsLatin1String` (*C fonksiyonu*), 106
- `PyUnicode_AsMBCSString` (*C fonksiyonu*), 107

`PyUnicode_AsRawUnicodeEscapeString` (*C fonksiyonu*), 105
`PyUnicode_AsUCS4` (*C fonksiyonu*), 97
`PyUnicode_AsUCS4Copy` (*C fonksiyonu*), 97
`PyUnicode_AsUnicode` (*C fonksiyonu*), 98
`PyUnicode_AsUnicodeAndSize` (*C fonksiyonu*), 98
`PyUnicode_AsUnicodeCopy` (*C fonksiyonu*), 98
`PyUnicode_AsUnicodeEscapeString` (*C fonksiyonu*), 105
`PyUnicode_AsUTF8` (*C fonksiyonu*), 102
`PyUnicode_AsUTF8AndSize` (*C fonksiyonu*), 102
`PyUnicode_AsUTF8String` (*C fonksiyonu*), 102
`PyUnicode_AsUTF16String` (*C fonksiyonu*), 104
`PyUnicode_AsUTF32String` (*C fonksiyonu*), 103
`PyUnicode_AsWideChar` (*C fonksiyonu*), 101
`PyUnicode_AsWideCharString` (*C fonksiyonu*), 101
`PyUnicode_Check` (*C fonksiyonu*), 92
`PyUnicode_CheckExact` (*C fonksiyonu*), 92
`PyUnicode_Compare` (*C fonksiyonu*), 109
`PyUnicode_CompareWithASCIIString` (*C fonksiyonu*), 109
`PyUnicode_Concat` (*C fonksiyonu*), 108
`PyUnicode_Contains` (*C fonksiyonu*), 109
`PyUnicode_CopyCharacters` (*C fonksiyonu*), 97
`PyUnicode_Count` (*C fonksiyonu*), 108
`PyUnicode_DATA` (*C fonksiyonu*), 92
`PyUnicode_Decode` (*C fonksiyonu*), 102
`PyUnicode_DecodeASCII` (*C fonksiyonu*), 106
`PyUnicode_DecodeCharmap` (*C fonksiyonu*), 106
`PyUnicode_DecodeFSDefault` (*C fonksiyonu*), 100
`PyUnicode_DecodeFSDefaultAndSize` (*C fonksiyonu*), 100
`PyUnicode_DecodeLatin1` (*C fonksiyonu*), 106
`PyUnicode_DecodeLocale` (*C fonksiyonu*), 99
`PyUnicode_DecodeLocaleAndSize` (*C fonksiyonu*), 99
`PyUnicode_DecodeMBCS` (*C fonksiyonu*), 107
`PyUnicode_DecodeMBCSStateful` (*C fonksiyonu*), 107
`PyUnicode_DecodeRawUnicodeEscape` (*C fonksiyonu*), 105
`PyUnicode_DecodeUnicodeEscape` (*C fonksiyonu*), 105
`PyUnicode_DecodeUTF7` (*C fonksiyonu*), 105
`PyUnicode_DecodeUTF7Stateful` (*C fonksiyonu*), 105
`PyUnicode_DecodeUTF8` (*C fonksiyonu*), 102
`PyUnicode_DecodeUTF8Stateful` (*C fonksiyonu*), 102
`PyUnicode_DecodeUTF16` (*C fonksiyonu*), 104
`PyUnicode_DecodeUTF16Stateful` (*C fonksiyonu*), 104
`PyUnicode_DecodeUTF32` (*C fonksiyonu*), 103
`PyUnicode_DecodeUTF32Stateful` (*C fonksiyonu*), 103
`PyUnicode_Encode` (*C fonksiyonu*), 102
`PyUnicode_EncodeASCII` (*C fonksiyonu*), 106
`PyUnicode_EncodeCharmap` (*C fonksiyonu*), 107
`PyUnicode_EncodeCodePage` (*C fonksiyonu*), 107
`PyUnicode_EncodeFSDefault` (*C fonksiyonu*), 100
`PyUnicode_EncodeLatin1` (*C fonksiyonu*), 106
`PyUnicode_EncodeLocale` (*C fonksiyonu*), 99
`PyUnicode_EncodeMBCS` (*C fonksiyonu*), 107
`PyUnicode_EncodeRawUnicodeEscape` (*C fonksiyonu*), 105
`PyUnicode_EncodeUnicodeEscape` (*C fonksiyonu*), 105
`PyUnicode_EncodeUTF7` (*C fonksiyonu*), 105
`PyUnicode_EncodeUTF8` (*C fonksiyonu*), 103
`PyUnicode_EncodeUTF16` (*C fonksiyonu*), 104
`PyUnicode_EncodeUTF32` (*C fonksiyonu*), 103
`PyUnicode_Fill` (*C fonksiyonu*), 97
`PyUnicode_Find` (*C fonksiyonu*), 108
`PyUnicode_FindChar` (*C fonksiyonu*), 108
`PyUnicode_Format` (*C fonksiyonu*), 109
`PyUnicode_FromEncodedObject` (*C fonksiyonu*), 96
`PyUnicode_FromFormat` (*C fonksiyonu*), 95
`PyUnicode_FromFormatV` (*C fonksiyonu*), 96
`PyUnicode_FromKindAndData` (*C fonksiyonu*), 95
`PyUnicode_FromObject` (*C fonksiyonu*), 99
`PyUnicode_FromString` (*C fonksiyonu*), 95
`PyUnicode_FromString()`, 113
`PyUnicode_FromStringAndSize` (*C fonksiyonu*), 95
`PyUnicode_FromUnicode` (*C fonksiyonu*), 98
`PyUnicode_FromWideChar` (*C fonksiyonu*), 101
`PyUnicode_FSConverter` (*C fonksiyonu*), 100
`PyUnicode_FSDecoder` (*C fonksiyonu*), 100
`PyUnicode_GET_DATA_SIZE` (*C fonksiyonu*), 93
`PyUnicode_GET_LENGTH` (*C fonksiyonu*), 92
`PyUnicode_GET_SIZE` (*C fonksiyonu*), 93
`PyUnicode_GetLength` (*C fonksiyonu*), 97
`PyUnicode_GetSize` (*C fonksiyonu*), 98
`PyUnicode_InternFromString` (*C fonksiyonu*), 109
`PyUnicode_InternInPlace` (*C fonksiyonu*), 109
`PyUnicode_IsIdentifier` (*C fonksiyonu*), 93
`PyUnicode_Join` (*C fonksiyonu*), 108
`PyUnicode_KIND` (*C fonksiyonu*), 92
`PyUnicode_MAX_CHAR_VALUE` (*C makrosu*), 93
`PyUnicode_New` (*C fonksiyonu*), 95
`PyUnicode_READ` (*C fonksiyonu*), 93
`PyUnicode_READ_CHAR` (*C fonksiyonu*), 93
`PyUnicode_ReadChar` (*C fonksiyonu*), 97
`PyUnicode_READY` (*C fonksiyonu*), 92
`PyUnicode_Replace` (*C fonksiyonu*), 108
`PyUnicode_RichCompare` (*C fonksiyonu*), 109
`PyUnicode_Split` (*C fonksiyonu*), 108
`PyUnicode_Splitlines` (*C fonksiyonu*), 108

- PyUnicode_Substring (*C fonksiyonu*), 97
- PyUnicode_Tailmatch (*C fonksiyonu*), 108
- PyUnicode_TransformDecimalToASCII (*C fonksiyonu*), 98
- PyUnicode_Translate (*C fonksiyonu*), 107
- PyUnicode_TranslateCharmap (*C fonksiyonu*), 107
- PyUnicode_Type (*C değişkeni*), 91
- PyUnicode_WCHAR_KIND (*C makrosu*), 92
- PyUnicode_WRITE (*C fonksiyonu*), 92
- PyUnicode_WriteChar (*C fonksiyonu*), 97
- PyUnicodeDecodeError_Create (*C fonksiyonu*), 30
- PyUnicodeDecodeError_GetEncoding (*C fonksiyonu*), 30
- PyUnicodeDecodeError_GetEnd (*C fonksiyonu*), 31
- PyUnicodeDecodeError_GetObject (*C fonksiyonu*), 30
- PyUnicodeDecodeError_GetReason (*C fonksiyonu*), 31
- PyUnicodeDecodeError_GetStart (*C fonksiyonu*), 30
- PyUnicodeDecodeError_SetEnd (*C fonksiyonu*), 31
- PyUnicodeDecodeError_SetReason (*C fonksiyonu*), 31
- PyUnicodeDecodeError_SetStart (*C fonksiyonu*), 30
- PyUnicodeEncodeError_Create (*C fonksiyonu*), 30
- PyUnicodeEncodeError_GetEncoding (*C fonksiyonu*), 30
- PyUnicodeEncodeError_GetEnd (*C fonksiyonu*), 31
- PyUnicodeEncodeError_GetObject (*C fonksiyonu*), 30
- PyUnicodeEncodeError_GetReason (*C fonksiyonu*), 31
- PyUnicodeEncodeError_GetStart (*C fonksiyonu*), 30
- PyUnicodeEncodeError_SetEnd (*C fonksiyonu*), 31
- PyUnicodeEncodeError_SetReason (*C fonksiyonu*), 31
- PyUnicodeEncodeError_SetStart (*C fonksiyonu*), 30
- PyUnicodeObject (*C tipi*), 91
- PyUnicodeTranslateError_Create (*C fonksiyonu*), 30
- PyUnicodeTranslateError_GetEnd (*C fonksiyonu*), 31
- PyUnicodeTranslateError_GetObject (*C fonksiyonu*), 30
- PyUnicodeTranslateError_GetReason (*C fonksiyonu*), 31
- PyUnicodeTranslateError_GetStart (*C fonksiyonu*), 30
- PyUnicodeTranslateError_SetEnd (*C fonksiyonu*), 31
- PyUnicodeTranslateError_SetReason (*C fonksiyonu*), 31
- PyUnicodeTranslateError_SetStart (*C fonksiyonu*), 30
- PyVarObject (*C tipi*), 184
- PyVarObject_HEAD_INIT (*C makrosu*), 185
- PyVarObject.ob_size (*C ögesi*), 196
- PyVectorcall_Call (*C fonksiyonu*), 62
- PyVectorcall_Function (*C fonksiyonu*), 62
- PyVectorcall_NARGS (*C fonksiyonu*), 62
- PyWeakref_Check (*C fonksiyonu*), 130
- PyWeakref_CheckProxy (*C fonksiyonu*), 130
- PyWeakref_CheckRef (*C fonksiyonu*), 130
- PyWeakref_GET_OBJECT (*C fonksiyonu*), 130
- PyWeakref_GetObject (*C fonksiyonu*), 130
- PyWeakref_NewProxy (*C fonksiyonu*), 130
- PyWeakref_NewRef (*C fonksiyonu*), 130
- PyWideStringList (*C tipi*), 160
- PyWideStringList_Append (*C fonksiyonu*), 160
- PyWideStringList_Insert (*C fonksiyonu*), 160
- PyWideStringList.items (*C ögesi*), 160
- PyWideStringList.length (*C ögesi*), 160
- PyWrapper_New (*C fonksiyonu*), 128
- ## R
- realloc(), 175
- referans sayısı, 235
- releasebufferproc (*C tipi*), 218
- repr
- gömülü fonksiyon, 59, 199
- reprfunc (*C tipi*), 217
- richcmpfunc (*C tipi*), 217
- ## S
- sanal makine, 237
- sanal ortam, 237
- sdterr
- stdin stdout, 143
- search
- path, module, 11, 142, 144
- sequence
- nesne, 88
- set
- nesne, 115
- set_all(), 8
- setattrfunc (*C tipi*), 217
- setattrofunc (*C tipi*), 217
- setswitchinterval() (*in module sys*), 146
- SIGINT, 29
- sınıf, 227
- sınıf değişkeni, 227
- SIZE_MAX, 85
- signal
- modülü, 29
- sihirli yöntem, 232
- soyut temel sınıf, 225

sözlük, [228](#)
sözlük anlama, [228](#)
sözlük görünümü, [228](#)
special
 method, [235](#)
ssizeargfunc (*C tipi*), [218](#)
ssizeobjargproc (*C tipi*), [218](#)
staticmethod
 gömülü fonksiyon, [187](#)
stderr (*in module sys*), [152](#)
stdin
 stdout stderr, [143](#)
stdin (*in module sys*), [152](#)
stdout
 stderr, stdin, [143](#)
stdout (*in module sys*), [152](#)
strerror(), [24](#)
string
 PyObject_Str (*C function*), [59](#)
sum_list(), [8](#)
sum_sequence(), [9](#), [10](#)
sürekli paketleme, [235](#)
sys
 modülü, [11](#), [142](#), [152](#)
SystemError (*built-in exception*), [121](#), [122](#)

T

tanımlayıcı, [228](#)
tek sevk, [235](#)
tercüman kapatma, [230](#)
ternaryfunc (*C tipi*), [218](#)
tip, [236](#)
tip takma adı, [236](#)
traverseproc (*C tipi*), [222](#)
tuple
 gömülü fonksiyon, [69](#), [113](#)
 nesne, [109](#)
tür ipucu, [236](#)
type
 gömülü fonksiyon, [60](#)
 nesne, [6](#), [79](#)

U

ULONG_MAX, [85](#)
unaryfunc (*C tipi*), [218](#)
uzatma modülü, [228](#)

Ü

üç tırnaklı dize, [236](#)

V

vectorcallfunc (*C tipi*), [61](#)
version (*in module sys*), [144](#), [145](#)
visitproc (*C tipi*), [221](#)

Y

yazı çözümleme, [235](#)

yazı dosyası, [236](#)
yeni stil sınıf, [233](#)
yıkabilir, [230](#)
yinelenebilir, [231](#)
yineleyici, [231](#)
yol benzeri nesne, [234](#)
yol giriş kancası, [234](#)
yol girişi, [234](#)
yol girişi bulucu, [234](#)
yol tabanlı bulucu, [234](#)
yorumlanmış, [230](#)
yükleyici, [232](#)

Z

zorlama, [227](#)