
The Python/C API

출시 버전 **3.8.18**

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

2월 08, 2024

1	Introduction	3
1.1	Coding standards	3
1.2	Include Files	4
1.3	Useful macros	4
1.4	Objects, Types and Reference Counts	6
1.5	Exceptions	9
1.6	Embedding Python	11
1.7	Debugging Builds	12
2	안정적인 응용 프로그램 바이너리 인터페이스	13
3	매우 고수준 계층	15
4	참조 횟수	21
5	예외 처리	23
5.1	인쇄와 지우기	23
5.2	예외 발생시키기	24
5.3	경고 발행하기	26
5.4	에러 표시기 조회하기	27
5.5	시그널 처리하기	29
5.6	예외 클래스	29
5.7	예외 객체	30
5.8	유니코드 예외 객체	30
5.9	재귀 제어	32
5.10	표준 예외	32
5.11	표준 경고 범주	34
6	유틸리티	35
6.1	운영 체제 유틸리티	35
6.2	시스템 함수	38
6.3	프로세스 제어	40
6.4	모듈 임포트 하기	40
6.5	데이터 마샬링 지원	44
6.6	인자 구문 분석과 값 구축	45
6.7	문자열 변환과 포매팅	52
6.8	리플렉션	54

6.9	코덱 등록소와 지원 함수	54
7	추상 객체 계층	57
7.1	객체 프로토콜	57
7.2	숫자 프로토콜	63
7.3	시퀀스 프로토콜	66
7.4	매핑 프로토콜	68
7.5	이터레이터 프로토콜	69
7.6	버퍼 프로토콜	69
7.7	납은 버퍼 프로토콜	76
8	구상 객체 계층	77
8.1	기본 객체	77
8.2	숫자 객체	80
8.3	시퀀스 객체	86
8.4	컨테이너 객체	112
8.5	함수 객체	117
8.6	기타 객체	120
9	초기화, 파이널리제이션 및 스레드	139
9.1	파이썬 초기화 전	139
9.2	전역 구성 변수	140
9.3	인터프리터 초기화와 파이널리제이션	142
9.4	프로세스 전체 매개 변수	143
9.5	스레드 상태와 전역 인터프리터 록	146
9.6	서브 인터프리터 지원	152
9.7	비동기 알림	153
9.8	프로파일링과 추적	154
9.9	고급 디버거 지원	155
9.10	스레드 로컬 저장소 지원	156
10	파이썬 초기화 구성	159
10.1	PyWideStringList	160
10.2	PyStatus	161
10.3	PyPreConfig	162
10.4	PyPreConfig를 사용한 사전 초기화	163
10.5	PyConfig	164
10.6	PyConfig를 사용한 초기화	168
10.7	격리된 구성	170
10.8	파이썬 구성	170
10.9	경로 구성	171
10.10	Py_RunMain()	172
10.11	다단계 초기화 비공개 잠정적 API	173
11	메모리 관리	175
11.1	개요	175
11.2	원시 메모리 인터페이스	176
11.3	메모리 인터페이스	177
11.4	객체 할당자	178
11.5	기본 메모리 할당자	179
11.6	메모리 할당자 사용자 정의	179
11.7	pymalloc 할당자	181
11.8	tracemalloc C API	182
11.9	예	182

12 객체 구현 지원	185
12.1 힙에 객체 할당하기	185
12.2 공통 객체 구조체	186
12.3 형 객체	190
12.4 숫자 객체 구조체	214
12.5 매핑 객체 구조체	217
12.6 시퀀스 객체 구조체	217
12.7 버퍼 객체 구조체	218
12.8 비동기 객체 구조체	219
12.9 슬롯 형 typedef	220
12.10 예	221
12.11 순환 가비지 수집 지원	224
13 API와 ABI 버전 붙이기	227
A 용어집	229
B 이 설명서에 관하여	243
B.1 파이썬 설명서의 공헌자들	243
C 역사와 라이선스	245
C.1 소프트웨어의 역사	245
C.2 파이썬에 액세스하거나 사용하기 위한 이용 약관	246
C.3 포함된 소프트웨어에 대한 라이선스 및 승인	250
D 저작권	263
색인	265

이 설명서는 확장 모듈을 작성하거나 파이썬을 내장하고자 하는 C와 C++ 프로그래머가 사용하는 API에 관해 설명합니다. 이 설명서와 쌍을 이루는 `extending-index` 는 확장 제작의 일반 원칙을 설명하지만, API 함수를 자세하게 설명하지는 않습니다.

CHAPTER 1

Introduction

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).

참고: 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 [인자 구분 분석과 값 구축](#) 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.

참고: 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 you do not expect to be reached. 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.

버전 3.7에 추가.

Py_ABS(x)

Return the absolute value of `x`.

버전 3.3에 추가.

Py_MIN(x, y)

Return the minimum value between x and y.

버전 3.3에 추가.

Py_MAX(x, y)

Return the maximum value between x and y.

버전 3.3에 추가.

Py_STRINGIFY(x)

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

버전 3.4에 추가.

Py_MEMBER_SIZE(type, member)

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

버전 3.6에 추가.

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; }`.

버전 3.4에 추가.

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);
```

버전 3.8에서 변경: 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 pysqlite_row_methods[] = {
    {"keys", (PyCFunction)pysqlite_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;
    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;
    }
}
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```

    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++) {
        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.

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);
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```

    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 */
    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.

안정적인 응용 프로그램 바이너리 인터페이스

관례에 따라, 파이썬의 C API는 모든 배포마다 변경될 것입니다. 대부분 변경은 소스 호환되며, 일반적으로 기존 API를 변경하거나 API를 제거하지 않고 API를 추가하기만 합니다 (일부 인터페이스는 먼저 폐지된 후에 제거됩니다).

아쉽게도, API 호환성은 ABI(바이너리 호환성)로 확장되지 않습니다. 그 이유는 기본적으로 구조체 정의가 진화하기 때문인데, 새로운 필드를 추가하거나 필드의 형을 바꾸면 API가 손상되지는 않지만, ABI가 손상될 수 있습니다. 결과적으로, 확장 모듈은 파이썬 배포마다 다시 컴파일해야 합니다 (영향을 받는 인터페이스가 사용되지 않는 경우 유닉스에서는 예외일 수 있습니다). 또한, 윈도우에서 확장 모듈은 특정 `pythonXY.dll`과 링크되고 최신 모듈과 링크하기 위해 다시 컴파일할 필요가 있습니다.

파이썬 3.2부터, API 일부가 안정적인 ABI를 보장하도록 선언되었습니다. 이 API(“제한된 API”라고 합니다)를 사용하고자 하는 확장 모듈은 `Py_LIMITED_API`를 정의해야 합니다. 그러면 인터프리터의 세부 정보는 확장 모듈에 숨겨집니다; 그 대가로, 재컴파일 없이 모든 3.x 버전($x \geq 2$)에서 작동하는 모듈이 빌드됩니다.

어떤 경우에는, 안정적인 ABI를 새로운 기능으로 확장해야 합니다. 이러한 새로운 API를 사용하고자 하는 확장 모듈은 지원하고자 하는 최소 파이썬 버전의 `Py_VERSION_HEX` 값([API와 ABI 버전 불이기 참조](#))으로 `Py_LIMITED_API`를 설정해야 합니다 (예를 들어, 파이썬 3.3의 경우 `0x03030000`). 이러한 모듈은 모든 후속 파이썬 배포에서 작동하지만, 이전 배포에서 (심볼 누락으로 인해) 로드하지 못합니다.

파이썬 3.2부터, 제한된 API에서 사용할 수 있는 함수 집합이 [PEP 384](#)에 문서로 만들어져 있습니다. C API 설명서에서, 제한된 API 일부가 아닌 API 요소는 “제한된 API 일부가 아닙니다.”로 표시됩니다.

매우 고수준 계층

이 장의 함수들은 파일이나 버퍼에 제공된 파이썬 소스 코드를 실행할 수 있도록 하지만, 인터프리터와 더 세밀한 방식으로 상호 작용하도록 하지는 않습니다.

이러한 함수 중 일부는 문법의 시작 기호를 매개 변수로 받아들입니다. 사용 가능한 시작 기호는 `Py_eval_input`, `Py_file_input` 및 `Py_single_input`입니다. 이것들은 이들을 매개 변수로 받아들이는 함수 뒤에 설명됩니다.

또한 이 함수 중 일부는 `FILE*` 매개 변수를 취합니다. 주의해서 다루어야 할 한 가지 문제는 다른 C 라이브러리의 `FILE` 구조체가 다르고, 호환되지 않을 수 있다는 것입니다. (적어도) 윈도우에서는, 동적으로 링크된 확장에서 실제로 다른 라이브러리를 사용할 수 있어서, `FILE*` 매개 변수가 파이썬 런타임이 사용하고 있는 것과 같은 라이브러리에서 만들어진 것이 확실할 때만 이러한 함수에 전달되도록 주의해야 합니다.

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

표준 인터프리터의 메인 프로그램. 이것은 파이썬을 내장하는 프로그램을 위해 제공됩니다. *argc*와 *argv* 매개 변수는 C 프로그램의 `main()` 함수에 전달되는 것과 정확히 일치하도록 준비해야 합니다 (사용자의 로케일에 따라 `wchar_t`로 변환됩니다). 인자 목록이 수정될 수 있음에 유의해야 합니다 (하지만 인자 목록이 가리키는 문자열의 내용은 수정되지 않습니다). 인터프리터가 정상적으로 (즉, 예외 없이) 종료되면 반환 값은 0, 예외로 인해 인터프리터가 종료되면 1, 매개 변수 목록이 유효한 파이썬 명령 줄을 나타내지 않으면 2가 됩니다.

처리되지 않은 `SystemExit`가 발생하면, 이 함수는 `Py_InspectFlag`가 설정되어 있지 않은 한 1을 반환하지 않고 프로세스를 종료함에 유의하십시오.

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

`Py_Main()`과 유사하지만 *argv*는 바이트 문자열의 배열입니다.

버전 3.8에 추가.

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

아래 `PyRun_AnyFileExFlags()`의 단순화된 인터페이스입니다. *closeit*은 0으로 *flags*는 NULL로 설정된 상태로 남겨둡니다.

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

아래 `PyRun_AnyFileExFlags()`의 단순화된 인터페이스입니다. 이것은 *closeit* 인자를 0으로 설정된 상태로 남겨둡니다.

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

아래 *PyRun_AnyFileExFlags* () 의 단순화된 인터페이스입니다. 이것은 *flags* 인자를 NULL로 설정된 상태로 남겨둡니다.

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

*fp*가 대화식 장치(콘솔이나 터미널 입력이나 유닉스 의사 터미널)와 연결된 파일을 가리키면, *PyRun_InteractiveLoop* ()의 값을 반환하고, 그렇지 않으면 *PyRun_SimpleFile* ()의 결과를 반환합니다. *filename*은 파일 시스템 인코딩(sys.getfilesystemencoding ())으로 디코딩됩니다. *filename*이 NULL이면, 이 함수는 파일명으로 "???"를 사용합니다.

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

아래 *PyRun_SimpleStringFlags* () 의 단순화된 인터페이스입니다. *PyCompilerFlags** 인자를 NULL로 설정된 상태로 남겨둡니다.

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

flags 인자에 따라 `__main__` 모듈에서 *command*에 있는 파이썬 소스 코드를 실행합니다. `__main__`이 존재하지 않으면 만듭니다. 성공하면 0을, 예외가 발생하면 -1을 반환합니다. 예러가 있으면, 예외 정보를 얻는 방법이 없습니다. *flags*의 의미는 아래를 참조하십시오.

처리되지 않은 SystemExit가 발생하면, 이 함수는 Py_InspectFlag가 설정되어 있지 않은 한 -1을 반환하지 않고 프로세스를 종료함에 유의하십시오.

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

아래 *PyRun_SimpleFileExFlags* () 의 단순화된 인터페이스입니다. *closeit*을 0으로, *flags*를 NULL로 설정된 상태로 남겨둡니다.

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

아래 *PyRun_SimpleFileExFlags* () 의 단순화된 인터페이스입니다. *flags*를 NULL로 설정된 상태로 남겨둡니다.

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

PyRun_SimpleStringFlags () 와 비슷하지만, 메모리에 있는 문자열 대신 *fp*에서 파이썬 소스 코드를 읽습니다. *filename*은 파일의 이름이어야 하며, 파일 시스템 인코딩(sys.getfilesystemencoding ())으로 디코딩됩니다. *closeit*이 참이면 *PyRun_SimpleFileExFlags* 가 반환하기 전에 파일이 닫힙니다.

참고: 윈도우에서, *fp*는 바이너리 모드로 열어야 합니다(예를 들어 `fopen(filename, "rb")`). 그렇지 않으면, 파이썬은 LF 줄 종료가 있는 스크립트 파일을 올바르게 처리하지 못할 수 있습니다.

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

아래 *PyRun_InteractiveOneFlags* () 의 단순화된 인터페이스입니다. *flags*를 NULL로 설정된 상태로 남겨둡니다.

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

flags 인자에 따라 대화식 장치와 연관된 파일에서 단일 문장을 읽고 실행합니다. `sys.ps1`과 `sys.ps2`를 사용하여 사용자에게 프롬프트 합니다. *filename*은 파일 시스템 인코딩(sys.getfilesystemencoding ())으로 디코딩됩니다.

입력이 성공적으로 실행될 때 0을, 예외가 있으면 -1을, 또는 구문 분석 예러가 있으면 파이썬의 일부로 배포된 `errcode.h` 인클루드 파일에 있는 예러 코드를 반환합니다. (`errcode.h`는 `Python.h`에서 인클루드하지 않기 때문에 필요하면 특별히 인클루드해야 함에 유의하십시오.)

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

아래 *PyRun_InteractiveLoopFlags* () 의 단순화된 인터페이스입니다. *flags*를 NULL로 설정된 상태로 남겨둡니다.

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

EOF에 도달할 때까지 대화식 장치와 연관된 파일에서 문장을 읽고 실행합니다. `sys.ps1`

과 `sys.ps2`를 사용하여 사용자에게 프롬프트 합니다. `filename`은 파일 시스템 인코딩(`sys.getfilesystemencoding()`)으로 디코딩됩니다. EOF에서 0을 반환하거나, 실패하면 음수를 반환합니다.

int (*PyOS_InputHook) (void)

프로토타입 `int func(void)` 인 함수를 가리키도록 설정할 수 있습니다. 이 함수는 파이썬의 인터프리터 프롬프트가 유힬 상태가 되고 터미널에서 사용자 입력을 기다리려고 할 때 호출됩니다. 반환 값은 무시됩니다. 이 훅을 재정의하는 것은 파이썬 소스 코드의 `Modules/_tkinter.c`에서 한 것처럼 인터프리터의 프롬프트를 다른 이벤트 루프와 통합하는 데 사용될 수 있습니다.

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

프로토타입 `char *func(FILE *stdin, FILE *stdout, char *prompt)` 인 함수를 가리키도록 설정하여, 인터프리터의 프롬프트에서 단일 입력 줄을 읽는 데 사용되는 기본 함수를 재정의할 수 있습니다. 이 함수는 NULL이 아니면 문자열 `prompt`를 출력한 다음 제공된 표준 입력 파일에서 입력 줄을 읽고 결과 문자열을 반환할 것이라고 기대됩니다. 예를 들어, `readline` 모듈은 이 훅을 설정하여 줄 편집과 탭 완성 기능을 제공합니다.

결과는 `PyMem_RawMalloc()` 이나 `PyMem_RawRealloc()` 으로 할당된 문자열 이거나, 예외가 발생했으면 NULL이어야 합니다.

버전 3.4에서 변경: 결과는 `PyMem_Malloc()` 이나 `PyMem_Realloc()` 으로 할당하는 대신, `PyMem_RawMalloc()` 이나 `PyMem_RawRealloc()` 으로 할당해야 합니다.

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

아래 `PyParser_SimpleParseStringFlagsFilename()` 의 단순화된 인터페이스입니다. `filename`을 NULL로, `flags`를 0으로 설정된 상태로 남겨둡니다.

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

아래 `PyParser_SimpleParseStringFlagsFilename()` 의 단순화된 인터페이스입니다. 이것은 `filename`을 NULL로 설정된 상태로 남겨둡니다.

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

`flags` 인자에 따라 시작 토큰 `start`를 사용하여 `str`에서 파이썬 소스 코드를 구문 분석합니다. 결과는 효율적으로 평가될 수 있는 코드 객체를 생성하는 데 사용될 수 있습니다. 코드 조각을 여러 번 평가해야 하는 경우에 유용합니다. `filename`은 파일 시스템 인코딩(`sys.getfilesystemencoding()`)으로 디코딩됩니다.

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

아래 `PyParser_SimpleParseFileFlags()` 의 단순화된 인터페이스입니다. `flags`를 0으로 설정된 상태로 남겨둡니다.

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

`PyParser_SimpleParseStringFlagsFilename()` 와 유사하지만, 파이썬 소스 코드는 메모리에 있는 문자열 대신 `fp`에서 읽습니다.

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

Return value: New reference. 아래 `PyRun_StringFlags()` 의 단순화된 인터페이스입니다. 이것은 `flags`를 NULL로 설정된 상태로 남겨둡니다.

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

Return value: New reference. `flags`로 지정된 컴파일러 플래그를 사용하여 `globals`와 `locals` 객체로 지정된 컨텍스트에서 `str`에서 파이썬 소스 코드를 실행합니다. `globals`는 딕셔너리이어야 합니다. `locals`는 매핑 프로토콜을 구현하는 모든 객체가 될 수 있습니다. 매개 변수 `start`는 소스 코드를 구문 분석하는 데 사용해야 하는 시작 토큰을 지정합니다.

코드를 실행한 결과를 파이썬 객체로 반환하거나, 예외가 발생하면 NULL을 반환합니다.

PyObject* PyRun_File (FILE *fp, const char *filename, int start, PyObject *globals, PyObject *locals)
Return value: New reference. 아래 `PyRun_FileExFlags()`의 단순화된 인터페이스입니다. `closeit`을 0으로, `flags`를 NULL로 설정된 상태로 남겨둡니다.

PyObject* PyRun_FileEx (FILE *fp, const char *filename, int start, PyObject *globals, PyObject *locals, int closeit)
Return value: New reference. 아래 `PyRun_FileExFlags()`의 단순화된 인터페이스입니다. `flags`를 NULL로 설정된 상태로 남겨둡니다.

PyObject* PyRun_FileFlags (FILE *fp, const char *filename, int start, PyObject *globals, PyObject *locals, PyCompilerFlags *flags)
Return value: New reference. 아래 `PyRun_FileExFlags()`의 단순화된 인터페이스입니다. `closeit`을 0으로 설정된 상태로 남겨둡니다.

PyObject* PyRun_FileExFlags (FILE *fp, const char *filename, int start, PyObject *globals, PyObject *locals, int closeit, PyCompilerFlags *flags)
Return value: New reference. `PyRun_StringFlags()`와 유사하지만, 파이썬 소스 코드는 메모리에 있는 문자열 대신 `fp`에서 읽습니다. `filename`은 파일명이어야 하며 파일 시스템 인코딩(`sys.getfilesystemencoding()`)으로 디코딩됩니다. `closeit`이 참이면 `PyRun_FileExFlags()`가 반환되기 전에 파일이 닫힙니다.

PyObject* Py_CompileString (const char *str, const char *filename, int start)
Return value: New reference. 아래 `Py_CompileStringFlags()`의 단순화된 인터페이스입니다. `flags`를 NULL로 설정된 상태로 유지합니다.

PyObject* Py_CompileStringFlags (const char *str, const char *filename, int start, PyCompilerFlags *flags)
Return value: New reference. 아래 `Py_CompileStringExFlags()`의 단순화된 인터페이스입니다. `optimize`를 -1로 설정된 상태로 유지합니다.

PyObject* Py_CompileStringObject (const char *str, PyObject *filename, int start, PyCompilerFlags *flags, int optimize)
Return value: New reference. `str`에 있는 파이썬 소스 코드를 구문 분석하고 컴파일하여 결과 코드 객체를 반환합니다. 시작 토큰은 `start`로 주어집니다; 이것은 컴파일될 수 있는 코드를 제한하는 데 사용될 수 있으며 `Py_eval_input`, `Py_file_input` 또는 `Py_single_input`이어야 합니다. `filename`으로 지정된 파일명은 코드 객체를 구성하는 데 사용되며 트레이스백이나 `SyntaxError` 예외 메시지에 나타날 수 있습니다. 코드를 구문 분석할 수 없거나 컴파일할 수 없으면 NULL을 반환합니다.

정수 `optimize`는 컴파일러의 최적화 수준을 지정합니다. -1 값은 -O 옵션으로 주어진 것처럼 인터프리터의 최적화 수준을 선택합니다. 명시적 수준은 0 (최적화 없음; `__debug__`가 참), 1 (어쨌든이 제거되고 `__debug__`가 거짓) 또는 2 (독스트링도 제거됩니다)입니다.

버전 3.4에 추가.

PyObject* Py_CompileStringExFlags (const char *str, const char *filename, int start, PyCompilerFlags *flags, int optimize)
Return value: New reference. `Py_CompileStringObject()`와 유사하지만, `filename`은 파일 시스템 인코딩(`os.fsdecode()`)으로 디코딩된 바이트 문자열입니다.

버전 3.2에 추가.

PyObject* PyEval_EvalCode (PyObject *co, PyObject *globals, PyObject *locals)
Return value: New reference. 이것은 코드 객체와 전역 변수 및 지역 변수만 있는, `PyEval_EvalCodeEx()`의 단순화된 인터페이스입니다. 다른 인자는 NULL로 설정됩니다.

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. 주어진 평가를 위한 특정 환경에서, 미리 컴파일된 코드 객체를 평가합니다. 이 환경은 전역 변수의 딕셔너리, 지역 변수의 매핑 객체, 인자의 배열, 키워드와 기본값, 키워드 전용 인자의 기본값 딕셔너리 및 셀의 클로저 튜플로 구성됩니다.

PyFrameObject

프레임 객체를 기술하는 데 사용되는 객체의 C 구조체. 이 형의 필드는 언제든지 변경될 수 있습니다.

PyObject* PyEval_EvalFrame (PyFrameObject *f)

Return value: *New reference.* 실행 프레임을 평가합니다. 이전 버전과의 호환성을 위한, `PyEval_EvalFrameEx()`의 단순화된 인터페이스입니다.

PyObject* PyEval_EvalFrameEx (PyFrameObject *f, int throwflag)

Return value: *New reference.* 이것은 파이썬 인터프리터의 메인, 꾸미지 않은 함수입니다. 실행 프레임 *f*와 연관된 코드 객체가 실행됩니다. 필요에 따라 바이트 코드를 해석하고 호출을 실행합니다. 추가 *throwflag* 매개 변수는 대체로 무시할 수 있습니다 - 참이면, 예외가 즉시 발생하도록 합니다; 제너레이터 객체의 `throw()` 메서드에 사용됩니다.

버전 3.4에서 변경: 이 함수는 이제 활성 예외를 조용히 버리지 않았는지 확인하도록 도와려고 디버그 어서션을 포함합니다.

int PyEval_MergeCompilerFlags (PyCompilerFlags *cf)

이 함수는 현재 평가 프레임의 플래그를 변경하고, 성공하면 참을, 실패하면 거짓을 반환합니다.

int Py_eval_input

격리된 표현식을 위한 파이썬 문법의 시작 기호; `Py_CompileString()`과 함께 사용합니다.

int Py_file_input

파일이나 다른 소스에서 읽은 문장의 시퀀스를 위한 파이썬 문법의 시작 기호; `Py_CompileString()`과 함께 사용합니다. 임의로 긴 파이썬 소스 코드를 컴파일할 때 사용하는 기호입니다.

int Py_single_input

단일 문장을 위한 파이썬 문법의 시작 기호; `Py_CompileString()`과 함께 사용합니다. 대화식 인터프리터 루프에 사용되는 기호입니다.

struct PyCompilerFlags

이것은 컴파일러 플래그를 담는 데 사용되는 구조체입니다. 코드가 컴파일되기만 하는 경우 `int flags`로 전달되고, 코드가 실행되는 경우 `PyCompilerFlags *flags`로 전달됩니다. 이 경우, `from __future__ import`는 *flags*를 수정할 수 있습니다.

`PyCompilerFlags *flags`가 NULL일 때마다, `cf_flags`는 0과 같다고 취급되며, `from __future__ import`로 인한 수정은 버립니다.

int cf_flags

컴파일러 플래그.

int cf_feature_version

*cf_feature_version*은 부 파이썬 버전입니다. `PY_MINOR_VERSION`으로 초기화되어야 합니다.

이 필드는 기본적으로 무시되며, `PyCF_ONLY_AST` 플래그가 *cf_flags*에 설정된 경우에만 사용됩니다.

버전 3.8에서 변경: *cf_feature_version* 필드를 추가했습니다.

int CO_FUTURE_DIVISION

*flags*에서 이 비트를 설정하면 [PEP 238](#)에 따라 나누기 연산자 `/`를 “실수 나누기(true division)”로 해석되도록 합니다.

참조 횟수

이 섹션의 매크로는 파이썬 객체의 참조 횟수를 관리하는 데 사용됩니다.

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

객체 *o*에 대한 참조 횟수를 늘립니다. 객체는 NULL 일 수 없습니다; NULL이 아닌지 확실하지 않으면, *Py_XINCREF* ()를 사용하십시오.

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

객체 *o*에 대한 참조 횟수를 늘립니다. 객체는 NULL 일 수 있습니다, 이때 매크로는 효과가 없습니다.

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

객체 *o*에 대한 참조 횟수를 감소시킵니다. 객체는 NULL 일 수 없습니다; NULL이 아닌지 확실하지 않으면, *Py_XDECREF* ()를 사용하십시오. 참조 횟수가 0이 되면, 객체 형의 할당 해제 함수 (반드시 NULL이 아니어야 합니다)가 호출됩니다.

경고: 할당 해제 함수는 임의의 파이썬 코드가 호출되도록 할 수 있습니다 (예를 들어, `__del__` () 메서드가 있는 클래스 인스턴스가 할당 해제될 때). 이러한 코드에서의 예외는 전파되지 않지만, 실행된 코드는 모든 파이썬 전역 변수에 자유롭게 액세스할 수 있습니다. 이것은 *Py_DECREF* ()가 호출되기 전에 전역 변수에서 도달할 수 있는 모든 객체가 일관성 있는 상태에 있어야 함을 뜻합니다. 예를 들어, 리스트에서 객체를 삭제하는 코드는 삭제된 객체에 대한 참조를 임시 변수에 복사하고, 리스트 데이터 구조를 갱신한 다음, 임시 변수에 대해 *Py_DECREF* ()를 호출해야 합니다.

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

객체 *o*에 대한 참조 횟수를 감소시킵니다. 객체는 NULL 일 수 있습니다, 이때 매크로는 효과가 없습니다; 그렇지 않으면 효과는 *Py_DECREF* ()와 같으며 같은 경고가 적용됩니다.

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

객체 *o*에 대한 참조 횟수를 감소시킵니다. 객체는 NULL 일 수 있습니다, 이때 매크로는 효과가 없습니다; 그렇지 않으면 인자도 NULL로 설정된다는 점을 제외하고는, 효과가 *Py_DECREF* ()와 같습니다. 매크로가 임시 변수를 신중하게 사용하고, 참조 횟수를 줄이기 전에 인자를 NULL로 설정하기 때문에, *Py_DECREF* ()에 대한 경고는 전달된 객체와 관련하여 적용되지 않습니다.

가비지 수집 중에 탐색 될 수 있는 객체의 참조 횟수를 감소시킬 때마다 이 매크로를 사용하는 것이 좋습니다.

다음 함수는 파이썬의 실행 시간 동적 내장을 위한 것입니다: `Py_IncRef(PyObject *o)`, `Py_DecRef(PyObject *o)`. 이것들은 단순히 `Py_XINCREF()`와 `Py_XDECREF()`의 노출된 함수 버전입니다.

다음 함수나 매크로는 인터프리터 코어에서만 사용할 수 있습니다: `_Py_Dealloc()`, `_Py_ForgetReference()`, `_Py_NewReference()` 및 전역 변수 `_Py_RefTotal`.

예외 처리

이 장에서 설명하는 함수를 사용하면 파이썬 예외를 처리하고 발생시킬 수 있습니다. 파이썬 예외 처리의 기본 사항을 이해하는 것이 중요합니다. POSIX `errno` 변수와 비슷하게 작동합니다: 발생한 마지막 에러에 대한 전역 표시기(스레드 당)가 있습니다. 대부분 C API 함수는 성공 시 이를 지우지 않지만, 실패 시 에러의 원인을 나타내도록 설정합니다. 대부분 C API 함수는 에러 표시기도 반환합니다, 일반적으로 포인터를 반환해야 하면 NULL, 정수를 반환하면 -1을 반환합니다(예외: `PyArg_*()` 함수는 성공하면 1을, 실패하면 0을 반환합니다).

구체적으로, 에러 표시기는 세 가지 객체 포인터로 구성됩니다: 예외 형, 예외 값 및 트레이스백 객체. 이러한 포인터들은 설정되지 않으면 NULL이 될 수 있습니다(하지만 일부 조합은 금지되어 있습니다, 예를 들어 예외 형이 NULL이면 NULL이 아닌 트레이스백을 가질 수 없습니다).

호출한 일부 함수가 실패하여 함수가 실패해야 할 때, 일반적으로 에러 표시기를 설정하지 않습니다; 호출된 함수가 이미 설정했습니다. 에러를 처리하고 예외를 지우거나 보유한 모든 리소스(가령 객체 참조나 메모리 할당)를 정리한 후 반환해야 할 책임이 있습니다; 에러를 처리할 준비가 되지 않았을 때 정상적으로 계속되지 않아야 합니다. 에러로 인해 반환하면, 호출자에게 에러가 설정되었음을 알리는 것이 중요합니다. 에러를 처리하지 않거나 신중하게 전파하지 않으면, 파이썬/C API에 대한 추가 호출이 의도한 대로 작동하지 않을 수 있으며 알 수 없는 방식으로 실패할 수 있습니다.

참고: 에러 표시기는 `sys.exc_info()`의 결과가 **아닙니다**. 전자는 아직 포착되지 않은(따라서 여전히 전파 중인) 예외에 해당하는 반면, 후자는 포착된 후(따라서 전파가 중단된) 예외를 반환합니다.

5.1 인쇄와 지우기

`void PyErr_Clear()`

에러 표시기를 지웁니다. 에러 표시기가 설정되어 있지 않으면 효과가 없습니다.

`void PyErr_PrintEx(int set_sys_last_vars)`

표준 트레이스백을 `sys.stderr`로 인쇄하고 에러 표시기를 지웁니다. 에러가 `SystemExit`가 아닌 한, 이 경우에는 트레이스백이 인쇄되지 않고 파이썬 프로세스는 `SystemExit` 인스턴스에 의해 지정된 에러 코드로 종료됩니다.

에러 표시기가 설정된 경우**에만** 이 함수를 호출하십시오. 그렇지 않으면 치명적인 에러가 발생합니다!

`set_sys_last_vars`가 0이 아니면, 변수 `sys.last_type`, `sys.last_value` 및 `sys.last_traceback`은 각각 인쇄되는 예외의 형, 값 및 트레이스백으로 설정됩니다.

void **PyErr_Print** ()

`PyErr_PrintEx(1)`의 별칭.

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

현재 예외와 *obj* 인자를 사용하여 `sys.unraisablehook()`을 호출합니다.

이 유틸리티 함수는 예외가 설정되었지만, 인터프리터가 실제로 예외를 발생시킬 수 없을 때 `sys.stderr`에 경고 메시지를 인쇄합니다. 예를 들어, `__del__()` 메서드에서 예외가 발생할 때 사용됩니다.

이 함수는 발생시킬 수 없는 예외가 발생한 문맥을 식별하는 단일 인자 *obj*로 호출됩니다. 가능하면, *obj*의 `repr`이 경고 메시지에 인쇄됩니다.

이 함수를 호출할 때 예외를 설정되어 있어야 합니다.

5.2 예외 발생시키기

이 함수들은 현재 스레드의 에러 표시기를 설정하는 데 도움이 됩니다. 편의를 위해, 이러한 함수 중 일부는 항상 `return` 문에서 사용할 `NULL` 포인터를 반환합니다.

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

이것은 에러 표시기를 설정하는 가장 일반적인 방법입니다. 첫 번째 인자는 예외 형을 지정합니다; 일반적으로 표준 예외 중 하나입니다, 예를 들어 `PyExc_RuntimeError`. 참조 횟수를 증가시킬 필요가 없습니다. 두 번째 인자는 에러 메시지입니다; 'utf-8'에서 디코딩됩니다.

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

이 함수는 `PyErr_SetString()`과 유사하지만, 예외의 “값”에 대해 임의의 파이썬 객체를 지정할 수 있습니다.

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

Return value: Always `NULL`. 이 함수는 에러 표시기를 설정하고 `NULL`을 반환합니다. *exception*은 파이썬 예외 클래스여야 합니다. *format*과 후속 매개 변수는 에러 메시지를 포맷하는 데 도움이 됩니다; `PyUnicode_FromFormat()`에서와 같은 의미와 값을 갖습니다. *format*은 ASCII 인코딩된 문자열입니다.

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

Return value: Always `NULL`. `PyErr_Format()`과 같지만, 가변 개수의 인자 대신 *va_list* 인자를 취합니다.

버전 3.5에 추가.

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

이것은 `PyErr_SetObject(type, Py_None)`의 줄임 표현입니다.

int **PyErr_BadArgument** ()

이것은 `PyErr_SetString(PyExc_TypeError, message)`의 줄임 표현입니다, 여기서 *message*는 잘못된 인자로 내장 연산이 호출되었음을 나타냅니다. 대부분 내부 용입니다.

*PyObject** **PyErr_NoMemory** ()

Return value: Always `NULL`. 이것은 `PyErr_SetNone(PyExc_MemoryError)`의 줄임 표현입니다; `NULL`을 반환해서 객체 할당 함수는 메모리가 부족할 때 `return PyErr_NoMemory();`라고 쓸 수 있습니다.

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

Return value: Always NULL. C 라이브러리 함수가 에러를 반환하고 C 변수 `errno`를 설정했을 때 예외를 발생시키는 편의 함수입니다. 첫 번째 항목이 정수 `errno` 값이고 두 번째 항목이 (`strerror()`에서 얻은) 해당 에러 메시지인 튜플 객체를 만든 다음, `PyErr_SetObject(type, object)`를 호출합니다. 유닉스에서, `errno` 값이 시스템 호출이 중단되었음을 나타내는 `EINTR`이면, `PyErr_CheckSignals()`를 호출하고 이것이 에러 표시기를 설정하면, 설정된 그대로 됩니다. 이 함수는 항상 NULL을 반환해서, 시스템 호출에 대한 래퍼 함수는 시스템 호출이 에러를 반환할 때 `return PyErr_SetFromErrno(type);`이라고 쓸 수 있습니다.

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

Return value: Always NULL. `PyErr_SetFromErrno()`와 유사하지만, `filenameObject`가 NULL이 아니면, `type`의 생성자에 세 번째 매개 변수로 전달된다는 추가 동작이 있습니다. `OSError` 예외의 경우, 예외 인스턴스의 `filename` 어트리뷰트를 정의하는 데 사용됩니다.

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

Return value: Always NULL. `PyErr_SetFromErrnoWithFilenameObject()`와 유사하지만, 두 개의 파일명을 취하는 함수가 실패할 때 에러를 발생시키기 위해 두 번째 파일명 객체를 취합니다.

버전 3.4에 추가.

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

Return value: Always NULL. `PyErr_SetFromErrnoWithFilenameObject()`와 유사하지만, 파일명이 C 문자열로 제공됩니다. `filename`은 파일 시스템 인코딩(`os.fsdecode()`)으로 디코딩됩니다.

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

Return value: Always NULL. WindowsError를 발생시키는 편의 함수입니다. 0의 `ierr`로 호출하면, `GetLastError()` 호출에서 반환된 에러 코드가 대신 사용됩니다. Win32 함수 `FormatMessage()`를 호출하여 `ierr`이나 `GetLastError()`가 제공하는 에러 코드의 윈도우 설명을 얻은 다음, 첫 번째 항목이 `ierr` 값이고 두 번째 항목이 (`FormatMessage()`에서 얻은) 해당 에러 메시지인 튜플 객체를 생성한 다음, `PyErr_SetObject(PyExc_WindowsError, object)`를 호출합니다. 이 함수는 항상 NULL을 반환합니다.

가용성: 윈도우.

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

Return value: Always NULL. `PyErr_SetFromWindowsError()`와 유사하며, 발생시킬 예외 형을 지정하는 추가 매개 변수가 있습니다.

가용성: 윈도우.

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

Return value: Always NULL. `PyErr_SetFromWindowsErrorWithFilenameObject()`와 유사하지만, 파일명이 C 문자열로 제공됩니다. `filename`은 파일 시스템 인코딩(`os.fsdecode()`)으로 디코딩됩니다.

가용성: 윈도우.

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

Return value: Always NULL. `PyErr_SetFromWindowsErrorWithFilenameObject()`와 유사하며, 발생시킬 예외 형을 지정하는 추가 매개 변수가 있습니다.

가용성: 윈도우.

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

Return value: Always NULL. `PyErr_SetExcFromWindowsErrorWithFilenameObject()`와 유사하지만, 두 번째 파일명 객체를 받아들입니다.

가용성: 윈도우.

버전 3.4에 추가.

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

Return value: Always NULL. *PyErr_SetFromWindowsErrWithFilename*()와 유사하며, 발생시킬 예외 형을 지정하는 추가 매개 변수가 있습니다.

가용성: 윈도우.

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

Return value: Always NULL. ImportError를 발생시키는 편의 함수입니다. msg는 예외의 메시지 문자열로 설정됩니다. 둘 다 NULL이 될 수 있는, name과 path는 각각 ImportError의 name과 path 어트리뷰트로 설정됩니다.

버전 3.3에 추가.

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

현재 예외에 대한 파일(file), 줄(line) 및 오프셋(offset) 정보를 설정합니다. 현재 예외가 SyntaxError가 아니면, 추가 어트리뷰트를 설정하여, 예외 인쇄 하위 시스템이 예외가 SyntaxError라고 생각하게 합니다.

버전 3.4에 추가.

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

PyErr_SyntaxLocationObject()와 비슷하지만, filename은 파일 시스템 인코딩(os.fsdecode())에서 디코딩되는 바이트 문자열입니다.

버전 3.2에 추가.

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

PyErr_SyntaxLocationEx()와 비슷하지만, col_offset 매개 변수는 생략됩니다.

void **PyErr_BadInternalCall** ()

이것은 PyErr_SetString(PyExc_SystemError, message)의 줄임 표현입니다. 여기서 message는 내부 연산(예를 들어 파이썬/C API 함수)이 잘못된 인자로 호출되었음을 나타냅니다. 대부분 내부 용입니다.

5.3 경고 발행하기

이 함수를 사용하여 C 코드에서 경고를 발행하십시오. 파이썬 warnings 모듈에서 내보낸 유사한 함수를 미러링합니다. 일반적으로 sys.stderr에 경고 메시지를 인쇄합니다; 그러나, 사용자가 경고를 예외로 전환하도록 지정했을 수도 있으며, 이 경우 예외가 발생합니다. 경고 장치의 문제로 인해 이 함수가 예외를 발생시키는 것도 가능합니다. 예외가 발생하지 않으면 반환 값은 0이고, 예외가 발생하면 -1입니다. (경고 메시지가 실제로 인쇄되는지나 예외의 이유를 확인할 수 없습니다; 이것은 의도적입니다.) 예외가 발생하면, 호출자는 정상적인 예외 처리를 수행해야 합니다(예를 들어, 소유한 참조를 *Py_DECREF*()하고 예외 값을 반환합니다).

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

경고 메시지를 발행합니다. category 인자는 경고 범주(아래를 참조하십시오)나 NULL입니다; message 인자는 UTF-8로 인코딩된 문자열입니다. stack_level은 스택 프레임 수를 제공하는 양수입니다; 해당 스택 프레임에서 현재 실행 중인 코드 줄에서 경고가 발생합니다. stack_level이 1이면 *PyErr_WarnEx*()를 호출하는 함수, 2는 그 위의 함수, 등등.

경고 범주는 PyExc_Warning의 서브 클래스여야 합니다. PyExc_Warning은 PyExc_Exception의 서브 클래스입니다; 기본 경고 범주는 PyExc_RuntimeWarning입니다. 표준 파이썬 경고 범주는 이름이 표준 경고 범주에 열거된 전역 변수로 제공됩니다.

경고 제어에 대한 자세한 내용은, warnings 모듈 설명서와 명령 줄 설명서에서 -W 옵션을 참조하십시오. 경고 제어를 위한 C API는 없습니다.

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

Return value: Always NULL. *PyErr_SetImportError()* 와 매우 비슷하지만, 이 함수는 발생시킬 ImportError의 서브 클래스를 지정할 수 있습니다.

버전 3.6에 추가.

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

모든 경고 어트리뷰트를 명시적으로 제어하면서 경고 메시지를 발행합니다. 이것은 파이썬 함수 warnings.warn_explicit()에 대한 간단한 래퍼입니다. 자세한 내용은 거기를 참조하십시오. 그곳에 설명된 기본 효과를 얻으려면 module과 registry 인자를 NULL로 설정해야 합니다.

버전 3.4에 추가.

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

message와 module이 UTF-8 인코딩된 문자열이고, filename은 파일 시스템 인코딩(os.fsdecode())으로 디코딩된다는 점을 제외하면 PyErr_WarnExplicitObject()와 유사합니다.

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

*PyErr_WarnEx()*와 유사한 함수지만, *PyUnicode_FromFormat()*을 사용하여 경고 메시지를 포맷합니다. format은 ASCII 인코딩된 문자열입니다.

버전 3.2에 추가.

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

*PyErr_WarnFormat()*과 유사한 함수지만, category는 ResourceWarning이고 source를 warnings.WarningMessage()로 전달합니다.

버전 3.6에 추가.

5.4 예외 표시기 조회하기

*PyObject** **PyErr_Occurred** ()

Return value: Borrowed reference. 예외 표시기가 설정되었는지 테스트합니다. 설정되었으면, 예외 type(PyErr_Set*()) 함수나 *PyErr_Restore()*에 대한 마지막 호출의 첫 번째 인자를 반환합니다. 설정되지 않았으면, NULL을 반환합니다. 여러분이 반환 값에 대한 참조를 소유하지 않아서, *Py_DECREF()* 할 필요가 없습니다.

참고: 반환 값을 특정 예외와 비교하지 마십시오; 대신 *PyErr_ExceptionMatches()*를 사용하십시오, 아래를 참조하십시오. (클래스 예외의 경우 예외가 클래스 대신 인스턴스이거나, 예상하는 예외의 서브 클래스일 수 있어서 비교는 실패하기 쉽습니다.)

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

*PyErr_GivenExceptionMatches(PyErr_Occurred(), exc)*와 동등합니다. 예외가 실제로 설정되었을 때만 호출해야 합니다; 예외가 발생하지 않았으면 메모리 액세스 위반이 발생합니다.

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

given 예외가 exc의 예외 형과 일치하면 참을 반환합니다. exc가 클래스 객체이면, given이 서브 클래스의 인스턴스일 때도 참을 반환합니다. exc가 튜플이면, 튜플에 있는 모든 예외 형(그리고 서브 튜플도 재귀적으로)을 일치하기 위해 검색합니다.

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

주소가 전달된 세 개의 변수로 에러 표시기를 꺼냅니다. 에러 표시기가 설정되지 않았으면, 세 변수를 모두 NULL로 설정합니다. 설정되었으면, 지워지고 꺼낸 각 객체에 대한 참조를 여러분이 소유합니다. 값과 트레이스백 객체는 형 객체가 그렇지 않을 때도 NULL일 수 있습니다.

참고: 이 함수는 일반적으로 예외를 포착해야 하는 코드나 에러 표시기를 일시적으로 저장하고 복원해야 하는 코드에서만 사용됩니다. 예를 들어:

```
{
    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*)

세 객체로 에러 표시기를 설정합니다. 에러 표시기가 이미 설정되어 있으면, 먼저 지워집니다. 객체가 NULL이면, 에러 표시기가 지워집니다. NULL type과 함께 NULL이 아닌 value나 traceback을 전달하지 마십시오. 예외 형은 클래스여야 합니다. 잘못된 예외 형이나 값을 전달하지 마십시오. (이러한 규칙을 위반하면 나중에 미묘한 문제가 발생합니다.) 이 호출은 각 객체에 대한 참조를 제거합니다: 호출 전에 각 객체에 대한 참조를 소유해야 하며 호출 후에는 더는 이러한 참조를 소유하지 않습니다. (이것을 이해할 수 없다면, 이 함수를 사용하지 마십시오. 경고했습니다.)

참고: 이 함수는 일반적으로 에러 표시기를 일시적으로 저장하고 복원해야 하는 코드에서만 사용됩니다. 현재 에러 표시기를 저장하려면 *PyErr_Fetch()*를 사용하십시오.

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

특정 상황에서, 아래의 *PyErr_Fetch()*가 반환하는 값은 “비 정규화”되었을 수 있습니다. 즉, *exc는 클래스 객체이지만 *val은 같은 클래스의 인스턴스가 아닙니다. 이 함수는 이 경우 클래스를 인스턴스화하는 데 사용할 수 있습니다. 값이 이미 정규화되어 있으면, 아무 일도 일어나지 않습니다. 지연된 정규화는 성능 향상을 위해 구현됩니다.

참고: 이 함수 예외 값에 `__traceback__` 어트리뷰트를 묵시적으로 설정하지 않습니다. 트레이스백을 적절하게 설정해야 하면, 다음과 같은 추가 스니펫이 필요합니다:

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

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

`sys.exc_info()`로 알려진 것과 같은, 예외 정보를 꺼냅니다. 이것은 새로 발생한 예외가 아니라, 이미 포착된 예외를 가리킵니다. 세 객체에 대한 새 참조를 반환합니다. 이 중 어느 것이든 NULL일 수 있습니다. 예외 정보 상태를 수정하지 않습니다.

참고: 이 함수는 일반적으로 예외를 처리하려는 코드에서 사용되지 않습니다. 오히려, 코드가 예외 상태를 임시로 저장하고 복원해야 할 때 사용할 수 있습니다. 예외 상태를 복원하거나 지우려면 *PyErr_SetExcInfo()*를 사용하십시오.

버전 3.3에 추가.

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

`sys.exc_info()` 로 알려진 것과 같은, 예외 정보를 설정합니다. 이것은 새로 발생한 예외가 아니라, 이미 포착된 예외를 가리킵니다. 이 함수는 인자의 참조를 훔칩니다. 예외 상태를 지우려면, 세 인자 모두에 NULL을 전달하십시오. 세 인자에 대한 일반적인 규칙은, `PyErr_Restore()` 를 참조하십시오.

참고: 이 함수는 일반적으로 예외를 처리하려는 코드에서 사용되지 않습니다. 오히려, 코드가 예외 상태를 임시로 저장하고 복원해야 할 때 사용할 수 있습니다. 예외 상태를 읽으려면 `PyErr_GetExcInfo()` 를 사용하십시오.

버전 3.3에 추가.

5.5 시그널 처리하기

int **PyErr_CheckSignals** ()

이 함수는 파이썬의 시그널 처리와 상호 작용합니다. 시그널이 프로세스로 전송되었는지 확인하고, 그렇다면, 해당 시그널 처리기를 호출합니다. `signal` 모듈이 지원되면, 파이썬으로 작성된 시그널 처리기를 호출할 수 있습니다. 모든 경우에, `SIGINT`의 기본 효과는 `KeyboardInterrupt` 예외를 발생시키는 것입니다. 예외가 발생하면 에러 표시기가 설정되고 함수는 -1을 반환합니다; 그렇지 않으면 함수는 0을 반환합니다. 에러 표시기는 이전에 설정한 경우 지워지거나 지워지지 않을 수 있습니다.

void **PyErr_SetInterrupt** ()

`SIGINT` 시그널 도착의 효과를 시뮬레이션합니다. 다음에 `PyErr_CheckSignals()` 가 호출되면, `SIGINT`에 대한 파이썬 시그널 처리기가 호출됩니다.

`SIGINT`가 파이썬에서 처리되지 않으면(`signal.SIG_DFL`이나 `signal.SIG_IGN`으로 설정되어서), 이 함수는 아무 작업도 수행하지 않습니다.

int **PySignal_SetWakeupFd** (int fd)

이 유틸리티 함수는 시그널이 수신될 때마다 시그널 번호가 단일 바이트로 기록되는 파일 기술자를 지정합니다. `fd`는 비 블로킹이어야 합니다. 이전의 파일 기술자를 반환합니다.

값 -1은 기능을 비활성화합니다; 이것이 초기 상태입니다. 이것은 파이썬의 `signal.set_wakeup_fd()` 와 동등하지만, 에러 검사는 없습니다. `fd`는 유효한 파일 기술자여야 합니다. 함수는 메인 스레드에서만 호출되어야 합니다.

버전 3.5에서 변경: 윈도우에서, 함수는 이제 소켓 핸들도 지원합니다.

5.6 예외 클래스

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

Return value: New reference. 이 유틸리티 함수는 새 예외 클래스를 만들고 반환합니다. `name` 인자는 새 예외의 이름, `module.classname` 형식의 C 문자열이어야 합니다. `base`와 `dict` 인자는 일반적으로 NULL입니다. 이렇게 하면 `Exception`(C에서 `PyExc_Exception`으로 액세스할 수 있습니다)에서 파생된 클래스 객체가 만들어집니다.

새 클래스의 `__module__` 어트리뷰트는 `name` 인자의 첫 번째 부분(마지막 점까지)으로 설정되고, 클래스 이름은 마지막 부분(마지막 점 뒤)으로 설정됩니다. `base` 인자는 대체 베이스 클래스를 지정하는 데 사용할 수 있습니다; 하나의 클래스나 클래스의 튜플일 수 있습니다. `dict` 인자는 클래스 변수와 메서드의 딕셔너리를 지정하는 데 사용할 수 있습니다.

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

Return value: New reference. 새로운 예외 클래스에 독스트링을 쉽게 부여할 수 있다는 점을 제외하면 `PyErr_NewException()`과 같습니다: `doc`이 NULL이 아니면, 예외 클래스에 대한 독스트링으로 사용 됩니다.

버전 3.2에 추가.

5.7 예외 객체

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

Return value: New reference. 파이썬에서 `__traceback__`을 통해 액세스 할 수 있는 새로운 참조로 예외와 관련된 트레이스백을 반환합니다. 관련된 트레이스백이 없으면, NULL을 반환합니다.

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

예외와 관련된 트레이스백을 `tb`로 설정합니다. 지우려면 `Py_None`을 사용하십시오.

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

Return value: New reference. 파이썬에서 `__context__`를 통해 액세스 할 수 있는 새 참조로 예외와 연관된 컨텍스트(다른 예외 인스턴스, 이것을 처리하는 도중 `ex`가 발생했습니다)를 반환합니다. 연결된 컨텍스트가 없으면 NULL을 반환합니다.

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

예외와 연관된 컨텍스트를 `ctx`로 설정합니다. 지우려면 NULL을 사용하십시오. `ctx`가 예외 인스턴스인지 확인하는 형 검사는 없습니다. 이것은 `ctx`에 대한 참조를 훔칩니다.

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

Return value: New reference. 파이썬에서 `__cause__`를 통해 액세스 할 수 있는 새 참조로 예외와 관련된 원인(예외 인스턴스나 `None`, `raise ... from ...`으로 설정됩니다)을 반환합니다.

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

예외와 관련된 원인을 `cause`로 설정합니다. 지우려면 NULL을 사용하십시오. `cause`가 예외 인스턴스나 `None`인지 확인하는 형 검사는 없습니다. 이것은 `cause`에 대한 참조를 훔칩니다.

`__suppress_context__`는 이 함수에 의해 묵시적으로 `True`로 설정됩니다.

5.8 유니코드 예외 객체

다음 함수는 C에서 유니코드 예외를 만들고 수정하는 데 사용됩니다.

*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. `encoding`, `object`, `length`, `start`, `end` 및 `reason` 어트리뷰트를 사용하여 `UnicodeDecodeError` 객체를 만듭니다. `encoding`과 `reason`은 UTF-8로 인코딩된 문자열입니다.

*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. `encoding`, `object`, `length`, `start`, `end` 및 `reason` 어트리뷰트를 사용하여 `UnicodeEncodeError` 객체를 만듭니다. `encoding`과 `reason`은 UTF-8로 인코딩된 문자열입니다.

버전 3.3부터 폐지: 3.11

`Py_UNICODE`는 파이썬 3.3부터 폐지되었습니다. `PyObject_CallFunction(PyExc_UnicodeEncodeError, "sOnns", ...)`로 마이그레이션 하십시오.

*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. *object*, *length*, *start*, *end* 및 *reason* 어트리뷰트를 사용하여 `UnicodeTranslateError` 객체를 만듭니다. *reason*은 UTF-8로 인코딩된 문자열입니다.

버전 3.3부터 폐지: 3.11

`Py_UNICODE`는 파이썬 3.3부터 폐지되었습니다. `PyObject_CallFunction(PyExc_UnicodeTranslateError, "Onns", ...)`로 마이그레이션 하십시오.

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

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

Return value: New reference. 주어진 예외 객체의 *encoding* 어트리뷰트를 반환합니다.

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

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

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

Return value: New reference. 주어진 예외 객체의 *object* 어트리뷰트를 반환합니다.

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*)

주어진 예외 객체의 *start* 어트리뷰트를 가져와서 **start*에 배치합니다. *start*는 NULL이 아니어야 합니다. 성공하면 0을, 실패하면 -1을 반환합니다.

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*)

주어진 예외 객체의 *start* 어트리뷰트를 *start*로 설정합니다. 성공하면 0을, 실패하면 -1을 반환합니다.

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*)

주어진 예외 객체의 *end* 어트리뷰트를 가져와서 **end*에 배치합니다. *end*는 NULL이 아니어야 합니다. 성공하면 0을, 실패하면 -1을 반환합니다.

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*)

주어진 예외 객체의 *end* 어트리뷰트를 *end*로 설정합니다. 성공하면 0을, 실패하면 -1을 반환합니다.

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

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

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

Return value: New reference. 주어진 예외 객체의 *reason* 어트리뷰트를 반환합니다.

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

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

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

주어진 예외 객체의 *reason* 어트리뷰트를 *reason*으로 설정합니다. 성공하면 0을, 실패하면 -1을 반환합니다.

5.9 재귀 제어

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).

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

재귀적 C 수준 호출이 막 수행되려고 하는 지점을 표시합니다.

USE_STACKCHECK가 정의되었으면, 이 함수는 `PyOS_CheckStack()` 을 사용하여 OS 스택이 오버플로되었는지 확인합니다. 이 경우, `MemoryError` 를 설정하고 0이 아닌 값을 반환합니다.

그런 다음 함수는 재귀 제한에 도달했는지 확인합니다. 이 경우, `RecursionError` 가 설정되고 0이 아닌 값이 반환됩니다. 그렇지 않으면, 0이 반환됩니다.

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

void **Py_LeaveRecursiveCall** ()

`Py_EnterRecursiveCall()` 을 종료합니다. `Py_EnterRecursiveCall()` 의 각 성공적인 호출마다 한 번씩 호출되어야 합니다.

컨테이너형에 대해 `tp_repr` 을 올바르게 구현하려면 특별한 재귀 처리가 필요합니다. 스택을 보호하는 것 외에도, `tp_repr` 은 순환을 방지하기 위해 객체를 추적해야 합니다. 다음 두 함수는 이 기능을 쉽게 만듭니다. 사실상, 이들은 `reprlib.recursive_repr()` 에 대한 C 동등물입니다.

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

순환을 감지하기 위해 `tp_repr` 구현 시작 시 호출됩니다.

객체가 이미 처리되었으면, 함수는 양의 정수를 반환합니다. 이 경우 `tp_repr` 구현은 순환을 나타내는 문자열 객체를 반환해야 합니다. 예를 들어, dict 객체는 {...}를 반환하고 list 객체는 [...]를 반환합니다.

재귀 제한에 도달하면 함수는 음의 정수를 반환합니다. 이 경우 `tp_repr` 구현은 일반적으로 NULL을 반환해야 합니다.

그렇지 않으면, 함수는 0을 반환하고 `tp_repr` 구현은 정상적으로 계속될 수 있습니다.

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

`Py_ReprEnter()` 를 종료합니다. 0을 반환하는 `Py_ReprEnter()` 호출마다 한 번씩 호출해야 합니다.

5.10 표준 예외

모든 표준 파이썬 예외는 `PyExc_` 뒤에 파이썬 예외 이름이 오는 이름의 전역 변수로 제공됩니다. *PyObject** 형입니다; 모두 클래스 객체입니다. 완전성을 위해, 다음은 모든 변수입니다:

C 이름	파이썬 이름	노트
<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>	

다음 페이지에 계속

표 1 - 이전 페이지에서 계속

C 이름	파이썬 이름	노트
PyExc_ChildProcessError	ChildProcessError	
PyExc_ConnectionAbortedError	ConnectionAbortedError	
PyExc_ConnectionError	ConnectionError	
PyExc_ConnectionRefusedError	ConnectionRefusedError	
PyExc_ConnectionResetError	ConnectionResetError	
PyExc_EOFError	EOFError	
PyExc_FileExistsError	FileExistsError	
PyExc_FileNotFoundError	FileNotFoundError	
PyExc_FloatingPointError	FloatingPointError	
PyExc_GeneratorExit	GeneratorExit	
PyExc_ImportError	ImportError	
PyExc_IndentationError	IndentationError	
PyExc_IndexError	IndexError	
PyExc_InterruptedError	InterruptedError	
PyExc_IsADirectoryError	IsADirectoryError	
PyExc_KeyError	KeyError	
PyExc_KeyboardInterrupt	KeyboardInterrupt	
PyExc_LookupError	LookupError	(1)
PyExc_MemoryError	MemoryError	
PyExc_ModuleNotFoundError	ModuleNotFoundError	
PyExc_NameError	NameError	
PyExc_NotADirectoryError	NotADirectoryError	
PyExc_NotImplementedError	NotImplementedError	
PyExc_OSError	OSError	(1)
PyExc_OverflowError	OverflowError	
PyExc_PermissionError	PermissionError	
PyExc_ProcessLookupError	ProcessLookupError	
PyExc_RecursionError	RecursionError	
PyExc_ReferenceError	ReferenceError	(2)
PyExc_RuntimeError	RuntimeError	
PyExc_StopAsyncIteration	StopAsyncIteration	
PyExc_StopIteration	StopIteration	
PyExc_SyntaxError	SyntaxError	
PyExc_SystemError	SystemError	
PyExc_SystemExit	SystemExit	
PyExc_TabError	TabError	
PyExc_TimeoutError	TimeoutError	
PyExc_TypeError	TypeError	
PyExc_UnboundLocalError	UnboundLocalError	
PyExc_UnicodeDecodeError	UnicodeDecodeError	
PyExc_UnicodeEncodeError	UnicodeEncodeError	
PyExc_UnicodeError	UnicodeError	
PyExc_UnicodeTranslateError	UnicodeTranslateError	
PyExc_ValueError	ValueError	
PyExc_ZeroDivisionError	ZeroDivisionError	

버전 3.3에 추가 : 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 및 PyExc_TimeoutError는 **PEP 3151**을 따라 도입되었습니다.

버전 3.5에 추가: PyExc_StopAsyncIteration 과 PyExc_RecursionError.

버전 3.6에 추가: PyExc_ModuleNotFoundError.

다음은 PyExc_OSError에 대한 호환성 별칭입니다:

C 이름	노트
PyExc_EnvironmentError	
PyExc_IOError	
PyExc_WindowsError	(3)

버전 3.3에서 변경: 이러한 별칭은 별도의 예외 형이었습니다.

노트:

- (1) 이것은 다른 표준 예외에 대한 베이스 클래스입니다.
- (2) 윈도우에서만 정의됩니다; 전 처리기 매크로 MS_WINDOWS가 정의되었는지 테스트하여 이를 사용하는 코드를 보호하십시오.

5.11 표준 경고 범주

모든 표준 파이썬 경고 범주는 PyExc_ 뒤에 파이썬 예외 이름이 오는 이름의 전역 변수로 제공됩니다. *PyObject** 형입니다; 모두 클래스 객체입니다. 완전성을 위해, 다음은 모든 변수입니다:

C 이름	파이썬 이름	노트
PyExc_Warning	Warning	(1)
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	

버전 3.2에 추가: PyExc_ResourceWarning.

노트:

- (1) 이것은 다른 표준 경고 범주의 베이스 클래스입니다.

이 장의 함수들은 C 코드의 플랫폼 간 호환성 개선, C에서 파이썬 모듈 사용, 함수 인자의 구문 분석 및 C 값으로부터 파이썬 값을 구성하는 것에 이르기까지 다양한 유틸리티 작업을 수행합니다.

6.1 운영 체제 유틸리티

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

Return value: New reference. path에 대한 파일 시스템 표현을 반환합니다. 객체가 str이나 bytes 객체이면, 참조 횟수가 증가합니다. 객체가 os.PathLike 인터페이스를 구현하면, __fspath__()가 str나 bytes 객체일 때 반환됩니다. 그렇지 않으면, TypeError가 발생하고 NULL이 반환됩니다.

버전 3.6에 추가.

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

이름이 filename인 표준 I/O 파일 fp를 대화식으로 간주하면 참(0이 아닙니다)을 반환합니다. isatty(fileno(fp))가 참인 파일의 경우입니다. 전역 플래그 Py_InteractiveFlag가 참이면, 이 함수는 filename 포인터가 NULL이거나 이름이 문자열 '<stdin>'이나 '???' 중 하나와 같을 때도 참을 반환합니다.

void **PyOS_BeforeFork** ()

프로세스 포크 전에 내부 상태를 준비하는 함수. fork() 나 현재 프로세스를 복제하는 유사한 함수를 호출하기 전에 호출해야 합니다. fork()가 정의된 시스템에서만 사용 가능합니다.

경고: C fork() 호출은 (“메인” 인터프리터의) “메인” 스레드에서만 이루어져야 합니다. PyOS_BeforeFork()도 마찬가지입니다.

버전 3.7에 추가.

void **PyOS_AfterFork_Parent** ()

프로세스 포크 후 일부 내부 상태를 갱신하는 함수. 프로세스 복제가 성공했는지와 관계없이, fork() 나 현재 프로세스를 복제하는 유사한 함수를 호출한 후 부모 프로세스에서 호출해야 합니다. fork()가 정의된 시스템에서만 사용 가능합니다.

경고: C `fork()` 호출은 (“메인” 인터프리터의) “메인” 스레드에서만 이루어져야 합니다. `PyOS_AfterFork_Parent()` 도 마찬가지입니다.

버전 3.7에 추가.

void **PyOS_AfterFork_Child()**

프로세스 포크 후 내부 인터프리터 상태를 갱신하는 함수. `fork()` 나 현재 프로세스를 복제하는 유사한 함수를 호출한 후, 프로세스가 파이썬 인터프리터를 다시 호출할 가능성이 있으면 자식 프로세스에서 호출해야 합니다. `fork()` 가 정의된 시스템에서만 사용 가능합니다.

경고: C `fork()` 호출은 (“메인” 인터프리터의) “메인” 스레드에서만 이루어져야 합니다. `PyOS_AfterFork_Child()` 도 마찬가지입니다.

버전 3.7에 추가.

더 보기:

`os.register_at_fork()` 를 사용하면 `PyOS_BeforeFork()`, `PyOS_AfterFork_Parent()` 및 `PyOS_AfterFork_Child()` 에서 호출될 사용자 정의 파이썬 함수를 등록 할 수 있습니다.

void **PyOS_AfterFork()**

프로세스 포크 후 일부 내부 상태를 갱신하는 함수; 파이썬 인터프리터가 계속 사용된다면 새로운 프로세스에서 호출되어야 합니다. 새 실행 파일이 새 프로세스에 로드되면, 이 함수를 호출할 필요가 없습니다.

버전 3.7부터 폐지: 이 함수는 `PyOS_AfterFork_Child()` 로 대체되었습니다.

int **PyOS_CheckStack()**

인터프리터에 스택 공간이 부족하면 참을 반환합니다. 이것은 신뢰성 있는 확인이지만, `USE_STACKCHECK`이 정의되어 있을 때만 사용할 수 있습니다 (현재 마이크로소프트 Visual C++ 컴파일러를 사용하는 윈도우에서). `USE_STACKCHECK`은 자동으로 정의됩니다; 여러분 자신의 코드에서 정의를 변경해서는 안 됩니다.

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

시그널 *i*에 대한 현재 시그널 처리기를 반환합니다. 이것은 `sigaction()` 이나 `signal()` 을 감싸는 얇은 래퍼입니다. 그 함수들을 직접 호출하지 마십시오! `PyOS_sighandler_t`는 `void (*) (int)`의 typedef 별칭입니다.

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

시그널 *i*의 시그널 처리기를 *h*로 설정합니다; 이전 시그널 처리기를 반환합니다. 이것은 `sigaction()` 이나 `signal()` 을 감싸는 얇은 래퍼입니다. 그 함수들을 직접 호출하지 마십시오! `PyOS_sighandler_t`는 `void (*) (int)`의 typedef 별칭입니다.

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

`surrogateescape` 에러 처리기를 사용하여 로케일 인코딩에서 바이트열을 디코딩합니다: 디코딩할 수 없는 바이트열은 `U+DC80..U+DCFF` 범위의 문자로 디코딩됩니다. 바이트 시퀀스를 서로게이트 문자로 디코딩할 수 있으면, 이들을 디코딩하는 대신 `surrogateescape` 에러 처리기를 사용하여 바이트열을 이스케이프합니다.

인코딩, 가장 높은 우선순위에서 가장 낮은 우선순위로:

- UTF-8, 맥 OS, 안드로이드 및 VxWorks에서;
- UTF-8, `Py_LegacyWindowsFSEncodingFlag`이 0일 때 윈도우에서;
- UTF-8, 파이썬 UTF-8 모드가 활성화되었을 때;
- ASCII, `LC_CTYPE` 로케일이 "C"이고, `nl_langinfo(CODESET)` 이 ASCII 인코딩(또는 별칭)을 반환하고, `mbstowcs()` 와 `wcstombs()` 함수가 ISO-8859-1 인코딩을 사용할 때.

- 현재 로케일 인코딩

새로 할당된 와이드 문자(wide character) 문자열에 대한 포인터를 반환합니다, 메모리를 해제하려면 `PyMem_RawFree()`를 사용하십시오. `size`가 NULL이 아니면, 널 문자를 제외한 와이드 문자 수를 `*size`에 기록합니다.

디코딩 에러나 메모리 할당 에러 시 NULL을 반환합니다. `size`가 NULL이 아니면, 메모리 에러 시 `*size`가 `(size_t)-1`로 설정되고, 디코딩 에러 시 `(size_t)-2`로 설정됩니다.

C 라이브러리에 버그가 없으면, 디코딩 에러가 발생하지 않아야 합니다.

문자열을 바이트열로 다시 인코딩하려면 `Py_EncodeLocale()` 함수를 사용하십시오.

더 보기:

`PyUnicode_DecodeFSDefaultAndSize()`와 `PyUnicode_DecodeLocaleAndSize()` 함수.

버전 3.5에 추가.

버전 3.7에서 변경: 이 함수는 이제 UTF-8 모드에서 UTF-8 인코딩을 사용합니다.

버전 3.8에서 변경: 이 함수는 이제 윈도우에서 `Py_LegacyWindowsFSEncodingFlag`가 0이면 UTF-8 인코딩을 사용합니다;

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

surrogateescape 에러 처리기를 사용하여 와이드 문자(wide character) 문자열을 로케일 인코딩으로 인코딩합니다: U+DC80..U+DCFF 범위의 서로게이트 문자는 바이트 0x80..0xFF로 변환됩니다.

인코딩, 가장 높은 우선순위에서 가장 낮은 우선순위로:

- UTF-8, 맥 OS, 안드로이드 및 VxWorks에서;
- UTF-8, `Py_LegacyWindowsFSEncodingFlag`이 0일 때 윈도우에서;
- UTF-8, 파이썬 UTF-8 모드가 활성화되었을 때;
- ASCII, LC_CTYPE 로케일이 "C"이고, `nl_langinfo(CODESET)`이 ASCII 인코딩(또는 별칭)을 반환하고, `mbstowcs()`와 `wcstombs()` 함수가 ISO-8859-1 인코딩을 사용할 때.
- 현재 로케일 인코딩

이 함수는 파이썬 UTF-8 모드에서 UTF-8 인코딩을 사용합니다.

새로 할당된 바이트열에 대한 포인터를 반환합니다, 메모리를 해제하려면 `PyMem_Free()`를 사용하십시오. 인코딩 에러나 메모리 할당 에러 시 NULL을 반환합니다

`error_pos`가 NULL이 아니면, `*error_pos`는 성공 시 `(size_t)-1`로 설정되고, 인코딩 에러 시 유효하지 않은 문자의 인덱스로 설정됩니다.

바이트열을 와이드 문자 문자열로 다시 디코딩하려면 `Py_DecodeLocale()` 함수를 사용하십시오.

더 보기:

`PyUnicode_EncodeFSDefault()`와 `PyUnicode_EncodeLocale()` 함수.

버전 3.5에 추가.

버전 3.7에서 변경: 이 함수는 이제 UTF-8 모드에서 UTF-8 인코딩을 사용합니다.

버전 3.8에서 변경: 이 함수는 이제 윈도우에서 `Py_LegacyWindowsFSEncodingFlag`가 0이면 UTF-8 인코딩을 사용합니다;

6.2 시스템 함수

`sys` 모듈의 기능을 C 코드에서 액세스 할 수 있게 하는 유틸리티 함수입니다. 모두 내부 스레드 상태 구조체에 포함된 현재 인터프리터 스레드의 `sys` 모듈의 디렉터리에 작동합니다.

`PyObject *PySys_GetObject` (const char *name)

Return value: Borrowed reference. `sys` 모듈에서 객체 `name`을 반환하거나, 존재하지 않으면 예외를 설정하지 않고 `NULL`을 반환합니다.

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

`v`가 `NULL`이 아닌 한 `sys` 모듈의 `name`을 `v`로 설정합니다. `NULL`이면 `name`은 `sys` 모듈에서 삭제됩니다. 성공하면 0, 에러 시 -1을 반환합니다.

`void PySys_ResetWarnOptions` ()

`sys.warnoptions`를 빈 리스트로 재설정합니다. 이 함수는 `Py_Initialize()` 이전에 호출할 수 있습니다.

`void PySys_AddWarnOption` (const wchar_t *s)

`s`를 `sys.warnoptions`에 추가합니다. 경고 필터 리스트에 영향을 주려면 `Py_Initialize()` 이전에 이 함수를 호출해야 합니다.

`void PySys_AddWarnOptionUnicode` (PyObject *unicode)

`unicode`를 `sys.warnoptions`에 추가합니다.

참고: 이 함수는 현재 CPython 구현 외부에서 사용할 수 없습니다. 효과가 있으려면 `Py_Initialize()`에서 `warnings`를 묵시적으로 임포트 하기 전에 호출해야 하지만, 유니코드 객체를 만들도록 허락할 수 있을 만큼 런타임이 충분히 초기화되기 전에는 호출할 수 없기 때문입니다.

`void PySys_SetPath` (const wchar_t *path)

`sys.path`를 플랫폼의 검색 경로 구분자(유닉스에서는 `:`, 윈도우에서는 `;`)로 구분된 경로 리스트여야 하는 `path`에서 찾은 경로의 리스트 객체로 설정합니다.

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

`format`으로 기술되는 출력 문자열을 `sys.stdout`에 기록합니다. 잘림이 발생하더라도 예외는 발생하지 않습니다(아래를 참조하십시오).

`format`은 포맷된 출력 문자열의 총 크기를 1000바이트 이하로 제한해야 합니다 – 1000바이트 이후에는, 출력 문자열이 잘립니다. 특히, 이것은 무제한 “%s” 포맷이 있어서는 안 됨을 의미합니다; “%.<N>s”를 사용하여 제한해야 합니다, 여기서 <N>은 <N>에 다른 포맷된 텍스트의 최대 크기를 더할 때 1000바이트를 초과하지 않도록 계산된 십진수입니다. 또한 “%f”도 주의하십시오, 아주 큰 숫자는 수백 자리를 인쇄할 수 있습니다.

문제가 발생하거나, `sys.stdout`가 설정되어 있지 않으면, 포맷된 메시지는 실제(C 수준) `stdout`에 기록됩니다.

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

`PySys_WriteStdout()`과 같지만, 대신 `sys.stderr`이나 `stderr`에 씁니다.

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

`PySys_WriteStdout()`과 유사한 함수이지만, 메시지를 `PyUnicode_FromFormatV()`를 사용하여 포맷하고 메시지를 임의의 길이로 자르지 않습니다.

버전 3.2에 추가.

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

`PySys_FormatStdout()`과 같지만, 대신 `sys.stderr`이나 `stderr`에 씁니다.

버전 3.2에 추가.

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

s를 -X 옵션 집합으로 구문 분석하고 `PySys_GetXOptions()`가 반환하는 현재 옵션 매핑에 추가합니다. 이 함수는 `Py_Initialize()` 이전에 호출할 수 있습니다.

버전 3.2에 추가.

PyObject ***PySys_GetXOptions** ()

Return value: Borrowed reference. `sys._xoptions`와 유사하게, -X 옵션의 현재 딕셔너리를 반환합니다. 에러가 발생하면, NULL이 반환되고 예외가 설정됩니다.

버전 3.2에 추가.

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

모든 활성 혹은 감사 이벤트를 발생시킵니다. 성공 시 0을 반환하고 실패 시 예외를 설정하여 0이 아닌 값을 반환합니다.

혹이 추가되었으면, *format*과 기타 인자를 사용하여 전달할 튜플을 구성합니다. N 외에도, `Py_BuildValue()`에서 사용된 것과 같은 포맷 문자를 사용할 수 있습니다. 빌드된 값이 튜플이 아니면, 단일 요소 튜플에 추가됩니다. (N 포맷 옵션은 참조를 소비하지만, 이 함수에 대한 인자가 소비될 지를 알 방법이 없기 때문에, 사용하면 참조 누수가 발생할 수 있습니다.)

PY_SSIZE_T_CLEAN이 정의되었는지와 관계없이, # 포맷 문자는 항상 `Py_ssize_t`로 처리되어야 합니다.

`sys.audit()`은 파이썬 코드와 동일한 기능을 수행합니다.

버전 3.8에 추가.

버전 3.8.2에서 변경: # 포맷 문자에 대해 `Py_ssize_t`를 요구합니다. 이전에는, 피할 수 없는 폐지 경고가 발생했습니다.

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

활성 감사 혹은 리스트에 콜러블 *hook*을 추가합니다. 성공하면 0을, 실패하면 0이 아닌 값을 반환합니다. 런타임이 초기화되었으면, 실패 시 에러도 설정합니다. 이 API를 통해 추가된 혹은 런타임이 만든 모든 인터프리터에 대해 호출됩니다.

userData 포인터는 *hook* 함수로 전달됩니다. *hook* 함수는 다른 런타임에서 호출될 수 있어서, 이 포인터는 파이썬 상태를 직접 참조하면 안 됩니다.

이 함수는 `Py_Initialize()` 이전에 호출해도 안전합니다. 런타임 초기화 후 호출되면, 기존 감사 혹은 알리고 `Exception`에서 서브 클래싱 된 에러를 발생 시켜 조용히 연산을 중단할 수 있습니다(다른 에러는 억제되지(silenced) 않습니다).

hook 함수는 `int (*)(const char *event, PyObject *args, void *userData)` 형입니다. 여기서 *args*는 `PyTupleObject` 임이 보장됩니다. *hook* 함수는 항상 이벤트를 발생시킨 파이썬 인터프리터가 GIL을 잡은 채로 호출됩니다.

감사에 대한 자세한 설명은 [PEP 578](#)을 참조하십시오. 이벤트를 발생시키는 런타임과 표준 라이브러리의 함수는 감사 이벤트 표에 나열되어 있습니다. 자세한 내용은 각 함수 설명서에 있습니다.

인자 없이 감사 이벤트 `sys.addaudithook`을 발생시킵니다.

버전 3.8에 추가.

6.3 프로세스 제어

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

치명적인 에러 메시지를 인쇄하고 프로세스를 죽입니다. 아무런 정리도 수행되지 않습니다. 이 함수는 파이썬 인터프리터를 계속 사용하는 것이 위험한 조건이 감지되었을 때만 호출해야 합니다; 예를 들어, 객체 관리가 손상된 것으로 보일 때. 유닉스에서는, 표준 C 라이브러리 함수 `abort()` 가 호출되어 core 파일을 생성하려고 시도합니다.

void **Py_Exit** (int status)

현재 프로세스를 종료합니다. 이것은 `Py_FinalizeEx()`를 호출한 다음 표준 C 라이브러리 함수 `exit(status)`를 호출합니다. `Py_FinalizeEx()`가 에러를 표시하면, 종료 상태는 120으로 설정됩니다.

버전 3.6에서 변경: 파이널리제이션에서의 에러가 더는 무시되지 않습니다.

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

`Py_FinalizeEx()`가 호출할 정리 함수를 등록합니다. 정리 함수는 인자 없이 호출되며 값을 반환하지 않아야 합니다. 최대 32개의 정리 함수를 등록할 수 있습니다. 등록이 성공하면, `Py_AtExit()`는 0을 반환합니다; 실패하면 -1을 반환합니다. 마지막에 등록된 정리 함수가 먼저 호출됩니다. 각 정리 함수는 최대 한 번 호출됩니다. 정리 함수 전에 파이썬의 내부 파이널리제이션이 완료되기 때문에, `func`에서 파이썬 API를 호출하면 안 됩니다.

6.4 모듈 임포트 하기

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

Return value: New reference. 이것은 아래 `PyImport_ImportModuleEx()`에 대한 단순화된 인터페이스입니다. `globals`와 `locals` 인자를 NULL로 설정하고 `level`은 0으로 설정합니다. `name` 인자에 점이 포함되면 (패키지의 서브 모듈을 지정할 때), `fromlist` 인자는 리스트 `['*']`로 설정해서 반환 값이 그렇지 않았을 때 반환되는 최상위 수준 패키지 대신에 이름 지정된 모듈이 되도록 합니다. (안타깝게도, `name`이 실제로 서브 모듈 대신 서브 패키지를 지정하면 추가적인 부작용이 발생합니다: 패키지의 `__all__` 변수에 지정된 서브 모듈들이 로드됩니다.) 임포트 한 모듈에 대한 새로운 참조를 반환하거나 실패 시 예외가 설정된 NULL을 반환합니다. 모듈을 임포트 하는 데 실패하면 `sys.modules`에 모듈을 남기지 않습니다.

이 함수는 항상 절대 임포트를 사용합니다.

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

Return value: New reference. 이 함수는 `PyImport_ImportModule()`의 폐지된 별칭입니다.

버전 3.3에서 변경: 이 기능은 다른 스레드가 임포트 잠금을 보유한 경우 즉시 실패했었습니다. 그러나 파이썬 3.3에서는, 잠금 방식이 대부분의 목적에서 모듈 단위 잠금으로 전환되었기 때문에, 이 함수의 특수한 동작은 더는 필요하지 않습니다.

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

Return value: New reference. 모듈을 임포트 합니다. 내장 파이썬 함수 `__import__()`를 통해 가장 잘 설명할 수 있습니다.

반환 값은 임포트 된 모듈이나 최상위 패키지에 대한 새로운 참조, 또는 실패 시 예외가 설정된 NULL입니다. `__import__()`와 마찬가지로, 비어 있지 않은 `fromlist`가 제공되지 않는 한, 패키지의 서브 모듈이 요청되었을 때의 반환 값은 최상위 패키지입니다.

임포트 실패는 `PyImport_ImportModule()`처럼 불완전한 모듈 객체를 제거합니다.

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

Return value: New reference. 모듈을 임포트 합니다. 표준 `__import__()` 함수가 이 함수를 직접 호출하기 때문에, 내장 파이썬 함수 `__import__()`를 통해 가장 잘 설명할 수 있습니다.

반환 값은 임포트 된 모듈이나 최상위 패키지에 대한 새로운 참조, 또는 실패 시 예외가 설정된 NULL입니다. `__import__()`와 마찬가지로, 비어 있지 않은 `fromlist`가 제공되지 않는 한, 패키지의 서브 모듈이 요청되었을 때의 반환 값은 최상위 패키지입니다.

버전 3.3에 추가.

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

Return value: New reference. `PyImport_ImportModuleLevelObject()`와 비슷하지만, `name`은 유니코드 객체 대신 UTF-8로 인코딩된 문자열입니다.

버전 3.3에서 변경: `level`의 음수 값은 더는 허용되지 않습니다.

PyObject* PyImport_Import (PyObject *name)

Return value: New reference. 이것은 현재 “임포트 혹은 함수”를 호출하는 고수준 인터페이스입니다 (명시적인 `level 0`을 사용하는데, 절대 임포트를 뜻합니다). 현재 전역의 `__builtins__`에 있는 `__import__()` 함수를 호출합니다. 이는 현재 환경에 설치된 임포트 혹은 사용하여 임포트가 수행됨을 의미합니다.

이 함수는 항상 절대 임포트를 사용합니다.

PyObject* PyImport_ReloadModule (PyObject *m)

Return value: New reference. 모듈을 다시 로드(reload) 합니다. 다시 로드된 모듈에 대한 참조를 반환하거나, 실패 시 예외가 설정된 NULL을 반환합니다 (이때 모듈은 여전히 존재합니다).

PyObject* PyImport_AddModuleObject (PyObject *name)

Return value: Borrowed reference. 모듈 이름에 해당하는 모듈 객체를 반환합니다. `name` 인자는 `package.module` 형식일 수 있습니다. 먼저 모듈 디렉터리 내에 있는지 확인하고, 없으면 새로 만들어 모듈 디렉터리에 삽입합니다. 실패 시 예외를 설정하고 NULL을 반환합니다.

참고: 이 함수는 모듈을 로드하거나 임포트 하지 않습니다; 모듈이 아직 로드되지 않았으면, 빈 모듈 객체를 얻게 됩니다. 모듈을 임포트 하려면 `PyImport_ImportModule()`이나 그 변형 중 하나를 사용하십시오. `name`에서 점으로 구분된 이름으로 암시된 패키지 구조는 이미 존재하지 않는다면 만들어지지 않습니다.

버전 3.3에 추가.

PyObject* PyImport_AddModule (const char *name)

Return value: Borrowed reference. `PyImport_AddModuleObject()`와 비슷하지만, `name`은 유니코드 객체 대신 UTF-8로 인코딩된 문자열입니다.

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

Return value: New reference. 주어진 모듈 이름 (`name`)(`package.module` 형식일 수 있습니다)과 파이썬 바이트 코드 파일에서 읽거나 내장 함수 `compile()`로 얻은 코드 객체로, 모듈을 로드합니다. 모듈 객체에 대한 새로운 참조를 반환하거나, 또는 예외가 발생하면 예외가 설정된 NULL을 반환합니다. 예외가 발생하면 `sys.modules`에서 `name`이 제거됩니다, `PyImport_ExecCodeModule()`에 진입할 때 `name`이 `sys.modules`에 이미 있어도 그렇습니다. `sys.modules`에 불완전하게 초기화된 모듈을 남겨두는 것은 위험합니다, 그러한 모듈을 임포트 할 때 모듈 객체가 알 수 없는 (그리고 아마도 모듈 작성자의 의도에 비추어볼 때 손상된) 상태에 있음을 알 방법이 없기 때문입니다.

모듈의 `__spec__`과 `__loader__`는 아직 설정되지 않았다면 적절한 값으로 설정됩니다. 스펙의 로더는 모듈의 `__loader__`(설정되었다면)로 설정되고, 그렇지 않으면 `SourceFileLoader`의 인스턴스로 설정됩니다.

모듈의 `__file__` 어트리뷰트는 코드 객체의 `co_filename`으로 설정됩니다. 해당한다면, `__cached__`도 설정됩니다.

이 함수는 이미 임포트 되었다면 모듈을 다시 로드합니다. 모듈을 다시 로드하는 의도된 방법은 `PyImport_ReloadModule()`을 참조하십시오.

`name`이 `package.module` 형식의 점으로 구분된 이름을 가리키면, 이미 만들어지지 않은 패키지 구조는 여전히 만들어지지 않습니다.

`PyImport_ExecCodeModuleEx()`와 `PyImport_ExecCodeModuleWithPathnames()`도 참조하십시오.

PyObject* PyImport_ExecCodeModuleEx (const char *name, PyObject *co, const char *pathname)
Return value: New reference. `PyImport_ExecCodeModule()`과 유사하지만, 모듈 객체의 `__file__` 어트리뷰트는 NULL이 아니라면 `pathname`으로 설정됩니다.

`PyImport_ExecCodeModuleWithPathnames()`도 참조하십시오.

PyObject* PyImport_ExecCodeModuleObject (PyObject *name, PyObject *co, PyObject *pathname, PyObject *cpathname)
Return value: New reference. `PyImport_ExecCodeModuleEx()`와 유사하지만, 모듈 객체의 `__cached__` 어트리뷰트는 NULL이 아니라면 `cpathname`으로 설정됩니다. 세 가지 함수 중 이것이 선호되는 것입니다.

버전 3.3에 추가.

PyObject* PyImport_ExecCodeModuleWithPathnames (const char *name, PyObject *co, const char *pathname, const char *cpathname)
Return value: New reference. `PyImport_ExecCodeModuleObject()`와 유사하지만, `name`, `pathname` 및 `cpathname`은 UTF-8로 인코딩된 문자열입니다. `pathname`의 값이 NULL로 설정된 경우 어떤 값이 `cpathname`에서 와야하는지 알아내려고 합니다.

버전 3.2에 추가.

버전 3.3에서 변경: 바이트 코드 경로만 제공되면 소스 경로를 계산할 때 `imp.source_from_cache()`를 사용합니다.

long PyImport_GetMagicNumber ()
 파이썬 바이트 코드 파일(일명 .pyc 파일)의 매직 번호(magic number)를 반환합니다. 매직 번호는 바이트 코드 파일의 처음 4바이트에 리틀 엔디안 바이트 순서로 존재해야 합니다. 예러 시 -1을 반환합니다.
 버전 3.3에서 변경: 실패 시 -1을 반환합니다.

const char* PyImport_GetMagicTag ()
PEP 3147 형식 파이썬 바이트 코드 파일 이름의 매직 태그 문자열을 반환합니다. `sys.implementation.cache_tag`의 값은 신뢰할 수 있고 이 함수 대신 사용해야 함에 유의하십시오.
 버전 3.2에 추가.

PyObject* PyImport_GetModuleDict ()
Return value: Borrowed reference. 모듈 관리에 사용되는 딕셔너리(일명 `sys.modules`)를 반환합니다. 이것은 인터프리터마다 존재하는 변수임에 유의하십시오.

PyObject* PyImport_GetModule (PyObject *name)
Return value: New reference. 주어진 이름으로 이미 임포트된 모듈을 반환합니다. 모듈이 아직 임포트되지 않았다면 NULL을 반환하지만 예러는 설정하지 않습니다. 조회에 실패하면 NULL을 반환하고 예러를 설정합니다.
 버전 3.7에 추가.

PyObject* PyImport_GetImporter (PyObject *path)
Return value: New reference. `sys.path/pkg.__path__` 항목 `path`를 위한 파인더 객체를 반환합니다. `sys.path_importer_cache` 딕셔너리에서 꺼낼 수도 있습니다. 아직 캐시되지 않았으면, 경로 항목을 처리할 수 있는 훅이 발견될 때까지 `sys.path_hooks`를 탐색합니다. 훅이 없으면 None을 반환합니다; 이것은 호출자에게 경로 기반 파인더가 이 경로 항목에 대한 파인더를 찾을 수 없음을 알려줍니다. `sys.path_importer_cache`에 결과를 캐시 합니다. 파인더 객체에 대한 새로운 참조를 반환합니다.

void _PyImport_Init ()
 임포트 메커니즘을 초기화합니다. 내부 전용입니다.

void **PyImport_Cleanup** ()

모듈 테이블을 비웁니다. 내부 전용입니다.

void **_PyImport_Fini** ()

임포트 메커니즘을 마무리합니다. 내부 전용입니다.

int **PyImport_ImportFrozenModule** (*PyObject* *name)

Return value: New reference. name이라는 이름의 프로즌 모듈(frozen module)을 로드합니다. 성공하면 1을, 모듈을 찾지 못하면 0을, 초기화에 실패하면 예외를 설정하고 -1을 반환합니다. 로드가 성공할 때 임포트된 모듈에 액세스하려면 *PyImport_ImportModule* ()을 사용하십시오. (잘못된 이름에 주의하십시오 — 이 함수는 모듈이 이미 임포트 되었을 때 다시 로드합니다.)

버전 3.3에 추가.

버전 3.4에서 변경: __file__ 어트리뷰트는 더는 모듈에 설정되지 않습니다.

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

PyImport_ImportFrozenModuleObject ()와 비슷하지만, name은 유니코드 객체 대신 UTF-8로 인코딩된 문자열입니다.

struct **_frozen**

이것은 **freeze** 유틸리티(파이썬 소스 배포의 Tools/freeze/를 참조하십시오)가 생성한 프로즌 모듈 디스크립터를 위한 구조체 형 정의입니다. Include/import.h에 있는 정의는 다음과 같습니다:

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

const struct *_frozen** **PyImport_FrozenModules**

이 포인터는 struct *_frozen* 레코드의 배열을 가리키도록 초기화되는데, 멤버가 모두 NULL이나 0인 레코드로 끝납니다. 프로즌 모듈이 임포트 될 때, 이 테이블에서 검색됩니다. 제삼자 코드는 이것을 사용하여 동적으로 생성된 프로즌 모듈 컬렉션을 제공할 수 있습니다.

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

기존의 내장 모듈 테이블에 단일 모듈을 추가합니다. 이것은 *PyImport_ExtendInittab* ()을 감싸는 편리한 래퍼인데, 테이블을 확장할 수 없으면 -1을 반환합니다. 새 모듈은 name이라는 이름으로 임포트 될 수 있으며, *initfunc* 함수를 처음 시도한 임포트에서 호출되는 초기화 함수로 사용합니다. *Py_Initialize* () 전에 호출해야 합니다.

struct **_inittab**

내장 모듈 목록에 있는 단일 항목을 기술하는 구조체. 각 구조체는 인터프리터에 내장된 모듈의 이름과 초기화 함수를 제공합니다. 이름은 ASCII로 인코딩된 문자열입니다. 파이썬을 내장하는 프로그램은 *PyImport_ExtendInittab* ()과 함께 이러한 구조체의 배열을 사용하여 추가 내장 모듈을 제공할 수 있습니다. 구조체는 Include/import.h에서 다음과 같이 정의됩니다:

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

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

내장 모듈 테이블에 모듈 컬렉션을 추가합니다. *newtab* 배열은 name 필드에 NULL을 포함하는 센티넬(sentinel) 항목으로 끝나야 합니다; 센티넬 값을 제공하지 않으면 메모리 오류가 발생할 수 있습니다. 성공하면 0을, 내부 테이블을 확장하기 위한 메모리가 충분하지 않으면 -1을 반환합니다. 실패하면, 내부 테이블에 모듈이 추가되지 않습니다. *Py_Initialize* () 전에 호출해야 합니다.

6.5 데이터 마샬링 지원

이러한 루틴은 C 코드가 `marshal` 모듈과 같은 데이터 형식을 사용하여 직렬화된 객체로 작업 할 수 있도록 합니다. 직렬화 형식으로 데이터를 쓰는 함수와 데이터를 다시 읽는 데 사용할 수 있는 추가 함수가 있습니다. 마샬링 된 데이터를 저장하는 데 사용되는 파일은 바이너리 모드로 열어야 합니다.

숫자 값은 최하위 바이트가 먼저 저장됩니다.

이 모듈은 두 가지 버전의 데이터 형식을 지원합니다: 버전 0은 역사적인 버전이고, 버전 1은 파일에서와 역마샬링할 때 인턴(intern) 된 문자열을 공유합니다. 버전 2는 부동 소수점 숫자에 대해 바이너리 형식을 사용합니다. `Py_MARSHAL_VERSION`은 현재 파일 형식을 나타냅니다 (현재 2).

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

long 정수 value를 file로 마샬합니다. value의 최하위 32비트 만 기록합니다; 기본 long 형의 크기와 관계없이. version은 파일 형식을 나타냅니다.

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)

파이썬 객체 value를 file로 마샬합니다. version은 파일 형식을 나타냅니다.

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. 마샬된 value 표현을 포함한 바이트열 객체를 반환합니다. version은 파일 형식을 나타냅니다.

다음 함수를 사용하면 마샬된 값을 다시 읽을 수 있습니다.

long PyMarshal_ReadLongFromFile (FILE *file)

읽기 위해 열린 FILE*의 데이터 스트림에서 C long을 반환합니다. 이 함수를 사용하면 long의 기본 크기와 관계없이 32비트 값만 읽을 수 있습니다.

예러 시, 적절한 예외(EOFError)를 설정하고 -1을 반환합니다.

int PyMarshal_ReadShortFromFile (FILE *file)

읽기 위해 열린 FILE*의 데이터 스트림에서 C short를 반환합니다. 이 함수를 사용하면 short의 기본 크기와 관계없이 16비트 값만 읽을 수 있습니다.

예러 시, 적절한 예외(EOFError)를 설정하고 -1을 반환합니다.

PyObject* PyMarshal_ReadObjectFromFile (FILE *file)

Return value: New reference. 읽기 위해 열린 FILE*의 데이터 스트림에서 파이썬 객체를 반환합니다.

예러 시, 적절한 예외(EOFError, ValueError 또는 TypeError)를 설정하고 NULL을 반환합니다.

PyObject* PyMarshal_ReadLastObjectFromFile (FILE *file)

Return value: New reference. 읽기 위해 열린 FILE*의 데이터 스트림에서 파이썬 객체를 반환합니다. `PyMarshal_ReadObjectFromFile()`와 달리, 이 함수는 더는 파일에서 객체를 읽지 않을 것이라고 가정함으로써, 파일 데이터를 메모리에 적극적으로 로드 할 수 있고, 파일에서 한 바이트씩 읽는 대신 메모리에 있는 데이터에서 역 직렬화가 작동할 수 있습니다. 파일에서 어떤 것도 읽지 않을 것이라는 확신이 들 경우에만 이 변형을 사용하십시오.

예러 시, 적절한 예외(EOFError, ValueError 또는 TypeError)를 설정하고 NULL을 반환합니다.

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

Return value: New reference. data가 가리키는 len 바이트를 포함하는 바이트 버퍼의 데이터 스트림에서 파이썬 객체를 반환합니다.

예러 시, 적절한 예외(EOFError, ValueError 또는 TypeError)를 설정하고 NULL을 반환합니다.

6.6 인자 구문 분석과 값 구축

이 함수들은 자체 확장 함수와 메서드를 만들 때 유용합니다. 추가 정보와 예제는 `extending-index`에 있습니다.

설명된 이러한 함수 중 처음 세 개인 `PyArg_ParseTuple()`, `PyArg_ParseTupleAndKeywords()` 및 `PyArg_Parse()`는 모두 예상 인자에 관한 사항을 함수에 알리는 데 사용되는 포맷 문자열(*format strings*)을 사용합니다. 포맷 문자열은 이러한 각 함수에 대해 같은 문법을 사용합니다.

6.6.1 인자 구문 분석

포맷 문자열은 0개 이상의 “포맷 단위(format units)”로 구성됩니다. 포맷 단위는 하나의 파이썬 객체를 설명합니다; 일반적으로 단일 문자나 괄호로 묶인 포맷 단위 시퀀스입니다. 몇 가지 예외를 제외하고, 괄호로 묶인 시퀀스가 아닌 포맷 단위는 일반적으로 이러한 함수에 대한 단일 주소 인자에 대응합니다. 다음 설명에서, 인용된(*quoted*) 형식은 포맷 단위입니다; (등근) 괄호 안의 항목은 포맷 단위와 일치하는 파이썬 객체 형입니다; [대괄호] 안의 항목은 주소를 전달해야 하는 C 변수의 형입니다.

문자열과 버퍼

이러한 포맷을 사용하면 연속적인 메모리 청크로 객체에 액세스 할 수 있습니다. 반환된 유니코드나 바이트열 영역에 대한 원시 저장소를 제공할 필요가 없습니다.

일반적으로, 포맷이 버퍼에 대한 포인터를 설정할 때, 버퍼는 해당 파이썬 객체에 의해 관리되고, 버퍼는 이 객체의 수명을 공유합니다. 여러분이 직접 메모리를 해제할 필요가 없습니다. 유일한 예외는 `es`, `es#`, `et` 및 `et#`입니다.

그러나, `Py_buffer` 구조체가 채워질 때, 하부 버퍼가 잠겨서, 호출자가 `Py_BEGIN_ALLOW_THREADS` 블록 내에서도 가변 데이터의 크기가 조정되거나 파괴될 위험 없이 이후에 버퍼를 사용할 수 있습니다. 결과적으로, 데이터 처리를 마친 후 (또는 모든 조기 중단인 경우) `PyBuffer_Release()`를 호출해야 합니다.

달리 명시되지 않는 한, 버퍼는 NUL로 종료되지 않습니다.

일부 포맷에는 읽기 전용 바이트열류 객체가 필요하며, 버퍼 구조체 대신 포인터를 설정합니다. 객체의 `PyBufferProcs.bf_releasebuffer` 필드가 NULL인지 확인하여 작동하며, 이때는 bytearray와 같은 가변 객체를 허용하지 않습니다.

참고: 모든 포맷의 # 변형(`s#`, `y#` 등)에 대해, 길이 인자의 형(`int`나 `Py_ssize_t`)은 `Python.h`를 포함하기 전에 매크로 `PY_SSIZE_T_CLEAN`을 정의하여 제어됩니다. 매크로가 정의되었으면, 길이는 `int`가 아닌 `Py_ssize_t`입니다. 이 동작은 향후 파이썬 버전에서 `Py_ssize_t`만 지원하고 `int` 지원을 중단하도록 변경됩니다. 항상 `PY_SSIZE_T_CLEAN`을 정의하는 것이 가장 좋습니다.

s (str) [const char *] 유니코드 객체를 문자열에 대한 C 포인터로 변환합니다. 기존 문자열에 대한 포인터는 여러분이 주소를 전달한 문자 포인터 변수에 저장됩니다. C 문자열은 NUL로 종료됩니다. 파이썬 문자열은 내장된 널 코드 포인트를 포함하지 않아야 합니다; 그렇다면 `ValueError` 예외가 발생합니다. 유니코드 객체는 'utf-8' 인코딩을 사용하여 C 문자열로 변환됩니다. 이 변환이 실패하면, `UnicodeError`가 발생합니다.

참고: 이 포맷은 바이트열류 객체를 받아들이지 않습니다. 파일 시스템 경로를 받아들이고 이를 C 문자열로 변환하려면, `PyUnicode_FSConverter()`를 `converter`로 O& 포맷을 사용하는 것이 좋습니다.

버전 3.5에서 변경: 이전에는, 파이썬 문자열에서 내장된 널 코드 포인트가 발견되면 `TypeError`가 발생했습니다.

s* (str 또는 바이트열류 객체) [Py_buffer] 이 포맷은 바이트열류 객체뿐만 아니라 유니코드 객체를 받아들입니다. 호출자가 제공한 *Py_buffer* 구조체를 채웁니다. 이 경우 결과 C 문자열은 내장된 NUL 바이트를 포함할 수 있습니다. 유니코드 객체는 'utf-8' 인코딩을 사용하여 C 문자열로 변환됩니다.

s# (str, 읽기 전용 바이트열류 객체) [const char *, int 또는 Py_ssize_t] 가변 객체를 받아들이지 않는다는 점을 제외하면, s*와 같습니다. 결과는 두 개의 C 변수에 저장됩니다. 첫 번째 변수는 C 문자열에 대한 포인터이고, 두 번째 변수는 길이입니다. 문자열은 내장 널 바이트를 포함할 수 있습니다. 유니코드 객체는 'utf-8' 인코딩을 사용하여 C 문자열로 변환됩니다.

z (str 또는 None) [const char *] s와 비슷하지만, 파이썬 객체가 None일 수도 있는데, 이 경우 C 포인터가 NULL로 설정됩니다.

z* (str, 바이트열류 객체 또는 None) [Py_buffer] s*와 비슷하지만, 파이썬 객체는 None일 수도 있습니다, 이 경우 *Py_buffer* 구조체의 buf 멤버가 NULL로 설정됩니다.

z# (str, 읽기 전용 바이트열류 객체 또는 None) [const char *, int 또는 Py_ssize_t] s#와 비슷하지만, 파이썬 객체는 None일 수도 있습니다, 이 경우 C 포인터가 NULL로 설정됩니다.

y (읽기 전용 바이트열류 객체) [const char *] 이 포맷은 바이트열류 객체를 문자열에 대한 C 포인터로 변환합니다; 유니코드 객체를 받아들이지 않습니다. 바이트열 버퍼는 내장 널 바이트를 포함하지 않아야 합니다; 만약 그렇다면, ValueError 예외가 발생합니다.

버전 3.5에서 변경: 이전에는, 바이트열 버퍼에서 내장 널 바이트가 발견되면 TypeError가 발생했습니다.

y* (바이트열류 객체) [Py_buffer] s*의 이 변형은 유니코드 객체가 아니라 바이트열류 객체만 받아들입니다. 바이너리 데이터를 받아들이는 권장 방법입니다.

y# (읽기 전용 바이트열류 객체) [const char *, int 또는 Py_ssize_t] s#의 이 변형은 유니코드 객체가 아니라 바이트열류 객체만 받아들입니다.

S (bytes) [PyBytesObject *] 변환을 시도하지 않고, 파이썬 객체가 bytes 객체일 것을 요구합니다. 객체가 바이트열 객체가 아니면 TypeError를 발생시킵니다. C 변수는 *PyObject**로 선언될 수도 있습니다.

Y (bytearray) [PyByteArrayObject *] 변환을 시도하지 않고, 파이썬 객체가 bytearray 객체일 것을 요구합니다. 객체가 bytearray 객체가 아니면 TypeError를 발생시킵니다. C 변수는 *PyObject**로 선언될 수도 있습니다.

u (str) [const Py_UNICODE *] 파이썬 유니코드 객체를 유니코드 문자의 NUL 종료 버퍼에 대한 C 포인터로 변환합니다. 기존 유니코드 버퍼에 대한 포인터로 채워질, *Py_UNICODE* 포인터 변수의 주소를 전달해야 합니다. *Py_UNICODE* 문자의 너비는 컴파일 옵션에 따라 다음에 유의하십시오 (16비트나 32비트입니다). 파이썬 문자열은 내장 널 코드 포인트를 포함하지 않아야 합니다; 만약 그렇다면, ValueError 예외가 발생합니다.

버전 3.5에서 변경: 이전에는, 파이썬 문자열에서 내장된 널 코드 포인트가 발견되면 TypeError가 발생했습니다.

Deprecated since version 3.3, will be removed in version 3.12: 이전 스타일 *Py_UNICODE* API의 일부입니다; *PyUnicode_AsWideCharString()* 을 사용하여 마이그레이션 하십시오.

u# (str) [const Py_UNICODE *, int 또는 Py_ssize_t] u의 이 변형은 두 개의 C 변수에 저장됩니다, 첫 번째 변수는 유니코드 데이터 버퍼에 대한 포인터이고, 두 번째 변수는 길이입니다. 이 변형은 널 코드 포인트를 허용합니다.

Deprecated since version 3.3, will be removed in version 3.12: 이전 스타일 *Py_UNICODE* API의 일부입니다; *PyUnicode_AsWideCharString()* 을 사용하여 마이그레이션 하십시오.

Z (str 또는 None) [const Py_UNICODE *] u와 비슷하지만, 파이썬 객체는 None일 수도 있습니다, 이 경우 *Py_UNICODE* 포인터가 NULL로 설정됩니다.

Deprecated since version 3.3, will be removed in version 3.12: 이전 스타일 *Py_UNICODE* API의 일부입니다; *PyUnicode_AsWideCharString()* 을 사용하여 마이그레이션 하십시오.

Z# (str 또는 None) [const Py_UNICODE *, int or Py_ssize_t] u#와 비슷하지만, 파이썬 객체는 None일 수도 있습니다, 이 경우 *Py_UNICODE* 포인터가 NULL로 설정됩니다.

Deprecated since version 3.3, will be removed in version 3.12: 이전 스타일 *Py_UNICODE* API의 일부입니다; *PyUnicode_AsWideCharString()* 을 사용하여 마이그레이션 하십시오.

U (str) [PyObject *] 변환을 시도하지 않고, 파이썬 객체가 유니코드 객체일 것을 요구합니다. 객체가 유니코드 객체가 아니면 *TypeError*를 발생시킵니다. C 변수는 *PyObject **로 선언될 수도 있습니다.

w* (읽기-쓰기 바이트열류 객체) [Py_buffer] 이 포맷은 읽기-쓰기 버퍼 인터페이스를 구현하는 모든 객체를 허용합니다. 호출자가 제공한 *Py_buffer* 구조체를 채웁니다. 버퍼에는 내장 널 바이트가 포함될 수 있습니다. 호출자는 버퍼로 할 일을 마치면 *PyBuffer_Release()*를 호출해야 합니다.

es (str) [const char *encoding, char **buffer] s의 이 변형은 유니코드를 문자 버퍼로 인코딩하는 데 사용됩니다. 내장 NUL 바이트가 포함되지 않은 인코딩된 데이터에 대해서만 작동합니다.

이 포맷에는 두 개의 인자가 필요합니다. 첫 번째는 입력으로만 사용되며, 인코딩 이름을 가리키는 NUL 종료 문자열로 *const char**이거나, 'utf-8' 인코딩이 사용되도록 하는 NULL이어야 합니다. 명명된 인코딩이 파이썬에 알려지지 않았으면 예외가 발생합니다. 두 번째 인자는 *char***여야 합니다; 참조하는 포인터의 값은 인자 텍스트의 내용이 있는 버퍼로 설정됩니다. 텍스트는 첫 번째 인자에 지정된 인코딩으로 인코딩됩니다.

*PyArg_ParseTuple()*은 필요한 크기의 버퍼를 할당하고, 인코딩된 데이터를 이 버퍼에 복사하고 새로 할당된 스토리지를 참조하도록 **buffer*를 조정합니다. 호출자는 사용 후에 할당된 버퍼를 해제하기 위해 *PyMem_Free()*를 호출해야 합니다.

et (str, bytes 또는 bytearray) [const char *encoding, char **buffer] 바이트 문자열 객체를 다시 코딩하지 않고 통과시킨다는 점을 제외하면 *es*와 같습니다. 대신, 구현은 바이트 문자열 객체가 매개 변수로 전달된 인코딩을 사용한다고 가정합니다.

es# (str) [const char *encoding, char **buffer, int 또는 Py_ssize_t *buffer_length] s#의 이 변형은 유니코드를 문자 버퍼로 인코딩하는 데 사용됩니다. *es* 포맷과 달리, 이 변형은 NUL 문자를 포함하는 입력 데이터를 허용합니다.

세 가지 인자가 필요합니다. 첫 번째는 입력으로만 사용되며, 인코딩 이름을 가리키는 NUL 종료 문자열로 *const char**이거나, 'utf-8' 인코딩이 사용되도록 하는 NULL이어야 합니다. 명명된 인코딩이 파이썬에 알려지지 않았으면 예외가 발생합니다. 두 번째 인자는 *char***여야 합니다; 참조하는 포인터의 값은 인자 텍스트의 내용이 있는 버퍼로 설정됩니다. 텍스트는 첫 번째 인자에 지정된 인코딩으로 인코딩됩니다. 세 번째 인자는 정수에 대한 포인터여야 합니다; 참조된 정수는 출력 버퍼의 바이트 수로 설정됩니다.

두 가지 작동 모드가 있습니다:

**buffer*가 NULL 포인터를 가리키면, 함수는 필요한 크기의 버퍼를 할당하고, 이 버퍼로 인코딩된 데이터를 복사하고 **buffer*를 새로 할당된 스토리지를 참조하도록 설정합니다. 호출자는 사용 후 할당된 버퍼를 해제하기 위해 *PyMem_Free()*를 호출해야 합니다.

**buffer*가 NULL이 아닌 포인터를 가리키면 (이미 할당된 버퍼), *PyArg_ParseTuple()*은 이 위치를 버퍼로 사용하고 **buffer_length*의 초깃값을 버퍼 크기로 해석합니다. 그런 다음 인코딩된 데이터를 버퍼에 복사하고 NUL 종료합니다. 버퍼가 충분히 크지 않으면, *ValueError*가 설정됩니다.

두 경우 모두, **buffer_length*는 후행 NUL 바이트를 제외한 인코딩된 데이터의 길이로 설정됩니다.

et# (str, bytes 또는 bytearray) [const char *encoding, char **buffer, int 또는 Py_ssize_t *buffer_length] 바이트 문자열 객체를 다시 코딩하지 않고 통과시킨다는 점을 제외하면 *es#*와 같습니다. 대신, 구현은 바이트 문자열 객체가 매개 변수로 전달된 인코딩을 사용한다고 가정합니다.

숫자

b(int) [unsigned char] 음이 아닌 파이썬 정수를 부호 없는 작은 정수로 변환하고, C unsigned char에 저장합니다.

B(int) [unsigned char] 오버플로 검사 없이 파이썬 정수를 작은 정수로 변환하고, C unsigned char에 저장합니다.

h(int) [short int] 파이썬 정수를 C short int로 변환합니다.

H(int) [unsigned short int] 오버플로 검사 없이, 파이썬 정수를 C unsigned short int로 변환합니다.

i(int) [int] 파이썬 정수를 일반 C int로 변환합니다.

I(int) [unsigned int] 오버플로 검사 없이, 파이썬 정수를 C unsigned int로 변환합니다.

l(int) [long int] 파이썬 정수를 C long int로 변환합니다.

k(int) [unsigned long] 오버플로 검사 없이 파이썬 정수를 C unsigned long으로 변환합니다.

L(int) [long long] 파이썬 정수를 C long long으로 변환합니다.

K(int) [unsigned long long] 오버플로 검사 없이 파이썬 정수를 C unsigned long long으로 변환합니다.

n(int) [Py_ssize_t] 파이썬 정수를 C Py_ssize_t로 변환합니다.

c(길이 1의 bytes 또는 bytearray) [char] 길이가 1인 bytes나 bytearray 객체로 표시된, 파이썬 바이트를 C char로 변환합니다.

버전 3.3에서 변경: bytearray 객체를 허용합니다.

C(길이 1의 str) [int] 길이가 1인 str 객체로 표시된, 파이썬 문자를 C int로 변환합니다.

f(float) [float] 파이썬 부동 소수점 숫자를 C float로 변환합니다.

d(float) [double] 파이썬 부동 소수점 숫자를 C double로 변환합니다.

D(complex) [Py_complex] 파이썬 복소수를 C Py_complex 구조체로 변환합니다.

기타 객체

O(object) [PyObject*] C 객체 포인터에 파이썬 객체를 (변환 없이) 저장합니다. 따라서 C 프로그램은 전달된 실제 객체를 받습니다. 객체의 참조 횟수는 증가하지 않습니다. 저장된 포인터는 NULL이 아닙니다.

O!(object) [typeobject, PyObject*] C 객체 포인터에 파이썬 객체를 저장합니다. 이것은 O와 유사하지만, 두 개의 C 인자를 취합니다: 첫 번째는 파이썬 형 객체의 주소이고, 두 번째는 객체 포인터가 저장되는 (PyObject* 형의) C 변수의 주소입니다. 파이썬 객체가 필요한 형이 아니면, TypeError가 발생합니다.

O&(object) [converter, anything] converter 함수를 통해 파이썬 객체를 C 변수로 변환합니다. 두 개의 인자를 취합니다: 첫 번째는 함수이고, 두 번째는 void *로 변환된, (임의의 형의) C 변수의 주소입니다. converter 함수는 다음과 같이 호출됩니다:

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

여기서 object는 변환할 파이썬 객체이고 address는 PyArg_Parse*() 함수에 전달된 void* 인자입니다. 반환된 status는 성공적인 변환의 경우 1이고 변환에 실패한 경우 0이어야 합니다. 변환이 실패하면, converter 함수는 예외를 발생시키고 address의 내용을 수정하지 않은 상태로 두어야 합니다.

converter가 Py_CLEANUP_SUPPORTED를 반환하면, 인자 구문 분석이 결국 실패하면 두 번째로 호출되어 변환기에 이미 할당된 메모리를 해제할 기회를 제공할 수 있습니다. 이 두 번째 호출에서, object 매개 변수는 NULL이 됩니다; address는 원래 호출과 같은 값을 갖습니다.

버전 3.1에서 변경: `Py_CLEANUP_SUPPORTED`가 추가되었습니다.

p (bool) [int] 전달된 값의 논리값을 테스트(불리언 **predicate**)하고 결과를 동등한 C 참/거짓 정숫값으로 변환합니다. 표현식이 참이면 int를 1로, 거짓이면 0으로 설정합니다. 모든 유효한 파이썬 값을 허용합니다. 파이썬이 논리값을 테스트하는 방법에 대한 자세한 내용은 `truth`를 참조하십시오.

버전 3.3에 추가.

(items) (tuple) [matching-items] 객체는 길이가 *items*에 있는 포맷 단위의 수인 파이썬 시퀀스여야 합니다. C 인자들은 *items*의 개별 포맷 단위에 대응해야 합니다. 시퀀스의 포맷 단위는 중첩될 수 있습니다.

“긴” 정수(값이 플랫폼의 `LONG_MAX`를 초과하는 정수)를 전달할 수 있지만 적절한 범위 검사가 수행되지 않습니다 — 수신 필드가 값을 수신하기에 너무 작을 때 최상위 비트가 자동으로 잘립니다(실제로, 이 의미는 C의 다운 캐스트에서 물려받았습니다 — 여러분의 경험은 다를 수 있습니다).

몇 가지 다른 문자는 포맷 문자열에서 의미가 있습니다. 중첩된 괄호 안에서는 나타날 수 없습니다. 그들은:

- | 파이썬 인자 리스트의 나머지 인자가 선택 사항임을 나타냅니다. 선택적 인자에 해당하는 C 변수는 기본값으로 초기화되어야 합니다 — 선택적 인자가 지정되지 않을 때, `PyArg_ParseTuple()`은 해당 C 변수의 내용을 건드리지 않습니다.

- \$ `PyArg_ParseTupleAndKeywords()` 전용: 파이썬 인자 리스트의 나머지 인자가 키워드 전용임을 나타냅니다. 현재, 모든 키워드 전용 인자는 선택적 인자여야 하므로, |는 항상 포맷 문자열에서 \$ 앞에 지정되어야 합니다.

버전 3.3에 추가.

- : 포맷 단위 리스트는 여기에서 끝납니다; 콜론 뒤의 문자열은 예러 메시지에서 함수 이름으로 사용됩니다 (`PyArg_ParseTuple()`이 발생시키는 예외의 “연관된 값”).

- ; 포맷 단위 리스트는 여기에서 끝납니다; 세미콜론 뒤의 문자열은 기본 예러 메시지의 예러 메시지 대신 예러 메시지로 사용됩니다. :와 ;는 서로를 배제합니다.

호출자에게 제공되는 모든 파이썬 객체 참조는 빌려온(*borrowed*) 참조임에 유의하십시오; 참조 횟수를 줄이지 마십시오!

이러한 함수에 전달되는 추가 인자는 포맷 문자열에 의해 형이 결정되는 변수의 주소여야 합니다; 이들은 입력 튜플의 값을 저장하는 데 사용됩니다. 위의 포맷 단위 리스트에서 설명된 대로, 이러한 매개 변수가 입력값으로 사용되는 몇 가지 경우가 있습니다; 이 경우 해당 포맷 단위에 대해 지정된 것과 일치해야 합니다.

변환이 성공하려면, *arg* 객체가 포맷과 일치해야 하며 포맷이 소진되어야 합니다. 성공하면, `PyArg_Parse*()` 함수는 참을 반환하고, 그렇지 않으면 거짓을 반환하고 적절한 예외를 발생시킵니다. 포맷 단위 중 하나의 변환 실패로 인해 `PyArg_Parse*()` 함수가 실패하면, 해당 주소의 변수와 그 뒤에 오는 포맷 단위는 건드리지 않습니다.

API 함수

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

위치 매개 변수만 지역 변수로 취하는 함수의 매개 변수를 구문 분석합니다. 성공하면 참을 반환합니다; 실패하면, 거짓을 반환하고 적절한 예외를 발생시킵니다.

int **PyArg_VaParse** (*PyObject* *args, const char *format, va_list vargs)

가변 개수의 인자가 아닌 *va_list*를 받아들인다는 점을 제외하면, `PyArg_ParseTuple()`과 동일합니다.

int **PyArg_ParseTupleAndKeywords** (*PyObject* *args, *PyObject* *kw, const char *format, char *key-words[], ...)

위치와 키워드 매개 변수를 모두 지역 변수로 취하는 함수의 매개 변수를 구문 분석합니다. *keywords* 인자는 키워드 매개 변수 이름의 NULL-종료 배열입니다. 빈 이름은 위치-전용 매개 변수를 나타냅니다. 성공하면 참을 반환합니다; 실패하면, 거짓을 반환하고 적절한 예외를 발생시킵니다.

버전 3.6에서 변경: 위치-전용 매개 변수에 대한 지원이 추가되었습니다.

int **PyArg_VaParseTupleAndKeywords** (*PyObject* *args, *PyObject* *kw, const char *format, char *key-words[], va_list vargs)

가변 개수의 인자가 아닌 va_list를 받아들인다는 점을 제외하면, *PyArg_ParseTupleAndKeywords()* 와 동일합니다.

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

키워드 인자 딕셔너리의 키가 문자열인지 확인합니다. *PyArg_ParseTupleAndKeywords()* 가 사용되지 않는 경우에만 필요합니다, 여기서는 이미 이 검사를 수행하기 때문입니다.

버전 3.2에 추가.

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

“이전 스타일” 함수의 인자 리스트를 분해하는 데 사용되는 함수 — 이들은 파이썬 3에서 제거된 METH_OLDARGS 매개 변수 구문 분석 메서드를 사용하는 함수입니다. 새 코드에서 매개 변수 구문 분석에 사용하는 것은 권장되지 않고, 표준 인터프리터에 있는 대부분의 코드는 더는 이런 목적으로 사용하지 않도록 수정되었습니다. 그러나, 다른 튜플을 분해하는 편리한 방법으로 남아 있으며, 그런 목적으로 계속 사용할 수 있습니다.

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

인자 형을 지정하기 위해 포맷 문자열을 사용하지 않는 더 간단한 형식의 매개 변수 조회. 이 방법으로 매개 변수를 꺼내는 함수는 함수나 메서드 테이블에서 *METH_VARARGS*로 선언되어야 합니다. 실제 매개 변수를 포함하는 튜플은 *args*로 전달되어야 합니다; 이것은 실제로 튜플이어야 합니다. 튜플의 길이는 *min* 이상 *max* 이하이어야 합니다; *min*과 *max*는 같을 수 있습니다. 추가 인자는 함수에 전달되어야 하며, 각 인자는 *PyObject* * 변수에 대한 포인터여야 합니다; 이들은 *args*의 값으로 채워집니다; 빌린 참조가 포함됩니다. *args*에서 제공하지 않는 선택적 매개 변수에 해당하는 변수는 채워지지 않습니다; 이것들은 호출자에 의해 초기화되어야 합니다. 이 함수는 성공하면 참을 반환하고 *args*가 튜플이 아니거나, 잘못된 수의 요소를 포함하면 거짓을 반환합니다; 실패하면 예외가 설정됩니다.

다음은 이 함수 사용의 예입니다, 약한 참조를 위한 *_weakref* 도우미 모듈의 소스에서 가져왔습니다:

```
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;
}
```

이 예제에서 *PyArg_UnpackTuple()* 에 대한 호출은 *PyArg_ParseTuple()* 에 대한 다음 호출과 전적으로 동등합니다:

```
PyArg_ParseTuple(args, "O|O:ref", &object, &callback)
```


6.6.2 값 구축

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

Return value: New reference. *PyArg_Parse*()* 함수 계열에서 받아들이는 것과 유사한 포맷 문자열과 값 시퀀스를 기반으로 새 값을 만듭니다. 값을 반환하거나 예외가 발생하면 NULL을 반환합니다; NULL이 반환되면 예외가 발생합니다.

*Py_BuildValue()*는 항상 튜플을 구축하지는 않습니다. 포맷 문자열에 둘 이상의 포맷 단위가 포함되었을 때만 튜플을 구축합니다. 포맷 문자열이 비어 있으면, None을 반환합니다; 정확히 하나의 포맷 단위를 포함하면, 해당 포맷 단위가 기술하는 객체가 무엇이건 반환합니다. 크기가 0이나 1인 튜플을 반환하도록 하려면, 포맷 문자열을 괄호로 묶으십시오.

s와 s# 포맷의 경우처럼, 메모리 버퍼가 데이터를 빌드 객체에 제공하기 위해 매개 변수로 전달될 때, 필요한 데이터가 복사됩니다. 호출자가 제공하는 버퍼는 *Py_BuildValue()*가 만든 객체에 의해 참조되지 않습니다. 즉, 여러분의 코드가 malloc()을 호출하고 할당된 메모리를 *Py_BuildValue()*에 전달하면, 인단 *Py_BuildValue()*가 반환되면 여러분이 코드가 해당 메모리에 대해 free()를 호출해야 합니다.

다음 설명에서, 인용된 형식은 포맷 단위입니다; (등근) 괄호 안의 항목은 포맷 단위가 반환할 파이썬 객체 형입니다; [대괄호] 안의 항목은 전달할 C 값의 형입니다.

문자 스페이스, 탭, 콜론 및 쉼표는 포맷 문자열에서 무시됩니다 (하지만 s#와 같은 포맷 단위 내에서는 아닙니다). 이것은 긴 포맷 문자열을 좀 더 읽기 쉽게 만드는 데 사용할 수 있습니다.

s (str 또는 None) [const char *] 'utf-8' 인코딩을 사용하여 널-종료 C 문자열을 파이썬 str 객체로 변환합니다. C 문자열 포인터가 NULL이면, None이 사용됩니다.

s# (str 또는 None) [const char *, int or Py_ssize_t] 'utf-8' 인코딩을 사용하여 C 문자열과 그 길이를 파이썬 str 객체로 변환합니다. C 문자열 포인터가 NULL이면, 길이가 무시되고 None이 반환됩니다.

y (bytes) [const char *] 이것은 C 문자열을 파이썬 bytes 객체로 변환합니다. C 문자열 포인터가 NULL이면, None이 반환됩니다.

y# (bytes) [const char *, int 또는 Py_ssize_t] 이것은 C 문자열과 그 길이를 파이썬 객체로 변환합니다. C 문자열 포인터가 NULL이면, None이 반환됩니다.

z (str 또는 None) [const char *] s와 같습니다.

z# (str 또는 None) [const char *, int 또는 Py_ssize_t] s#과 같습니다.

u (str) [const wchar_t *] 유니코드 (UTF-16 또는 UCS-4) 데이터의 널-종료 wchar_t 버퍼를 파이썬 유니코드 객체로 변환합니다. 유니코드 버퍼 포인터가 NULL이면, None이 반환됩니다.

u# (str) [const wchar_t *, int 또는 Py_ssize_t] 유니코드 (UTF-16 또는 UCS-4) 데이터 버퍼와 그 길이를 파이썬 유니코드 객체로 변환합니다. 유니코드 버퍼 포인터가 NULL이면, 길이가 무시되고 None이 반환됩니다.

U (str 또는 None) [const char *] s와 같습니다.

U# (str 또는 None) [const char *, int 또는 Py_ssize_t] s#과 같습니다.

i (int) [int] 일반 C int를 파이썬 정수 객체로 변환합니다.

b (int) [char] 일반 C char을 파이썬 정수 객체로 변환합니다.

h (int) [short int] 일반 C short int를 파이썬 정수 객체로 변환합니다.

l (int) [long int] C long int를 파이썬 정수 객체로 변환합니다.

B (int) [unsigned char] C unsigned char을 파이썬 정수 객체로 변환합니다.

H (int) [unsigned short int] C unsigned short int를 파이썬 정수 객체로 변환합니다.

I (int) [unsigned int] C unsigned int를 파이썬 정수 객체로 변환합니다.

k (int) [unsigned long] C unsigned long을 파이썬 정수 객체로 변환합니다.

L (int) [long long] C long long을 파이썬 정수 객체로 변환합니다.

K (int) [unsigned long long] C unsigned long long을 파이썬 정수 객체로 변환합니다.

n (int) [Py_ssize_t] C Py_ssize_t를 파이썬 정수로 변환합니다.

c (길이 1의 bytes) [char] 바이트를 나타내는 C int를 길이 1의 파이썬 bytes 객체로 변환합니다.

C (길이 1의 str) [int] 문자를 나타내는 C int를 길이 1의 파이썬 str 객체로 변환합니다.

d (float) [double] C double을 파이썬 부동 소수점 숫자로 변환합니다.

f (float) [float] C float를 파이썬 부동 소수점 숫자로 변환합니다.

D (complex) [Py_complex *] C *Py_complex* 구조체를 파이썬 복소수로 변환합니다.

O (object) [PyObject *] 파이썬 객체를 손대지 않고 전달합니다 (1 증가하는 참조 횟수는 예외입니다). 전달된 객체가 NULL 포인터면, 인자를 생성하는 호출이 에러를 발견하고 예외를 설정했기 때문으로 간주합니다. 따라서, *Py_BuildValue()*는 NULL을 반환하지만, 예외를 발생시키지 않습니다. 아직 예외가 발생하지 않았으면, *SystemError*가 설정됩니다.

S (object) [PyObject *] O와 같습니다.

N (object) [PyObject *] O와 같지만, 객체의 참조 횟수를 증가시키지 않습니다. 인자 리스트에서 객체 생성자를 호출하여 객체를 만들 때 유용합니다.

O& (object) [converter, anything] *converter* 함수를 통해 *anything*을 파이썬 객체로 변환합니다. 함수는 인자로 *anything*(void*와 호환되어야 합니다)을 사용하여 호출되며 “새” 파이썬 객체를 반환하거나, 에러가 발생하면 NULL을 반환해야 합니다.

(items) (tuple) [matching-items] C 값의 시퀀스를 항목 수가 같은 파이썬 튜플로 변환합니다.

[items] (list) [matching-items] C 값의 시퀀스를 항목 수가 같은 파이썬 리스트로 변환합니다.

{items} (dict) [matching-items] C 값의 시퀀스를 파이썬 딕셔너리로 변환합니다. 연속된 C 값의 각 쌍은 딕셔너리에 하나의 항목을 추가하여, 각각 키와 값으로 사용됩니다.

포맷 문자열에 에러가 있으면, *SystemError* 예외가 설정되고 NULL이 반환됩니다.

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

Return value: New reference. 가변 개수의 인자가 아닌 va_list를 받아들인다는 점을 제외하면, *Py_BuildValue()*와 동일합니다.

6.7 문자열 변환과 포매팅

숫자 변환과 포맷된 문자열 출력을 위한 함수.

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()*와 *PyOS_vsnprintf()*는 표준 C 라이브러리 함수 *snprintf()*와 *vsnprintf()*를 감쌉니다. 그들의 목적은 경계 조건에서 표준 C 함수가 제공하지 않는 수준의 일관된 동작을 보장하는 것입니다.

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`.

플랫폼에 `vsnprintf()`가 없고 잘림을 방지하는 데 필요한 버퍼 크기가 *size*를 512바이트보다 더 초과하면, 파이썬은 `Py_FatalError()`로 중단됩니다.

이 함수들의 반환 값(*rv*)은 다음과 같이 해석되어야 합니다:

- 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.

다음 함수는 로케일 독립적인 문자열에서 숫자로의 변환을 제공합니다.

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

문자열 *s*를 `double`로 변환하고, 실패 시 파이썬 예외를 발생시킵니다. 허용되는 문자열 집합은 *s*가 선행이나 후행 공백을 가질 수 없다는 점을 제외하고는 파이썬의 `float()` 생성자가 허용하는 문자열 집합에 대응합니다. 변환은 현재 로케일과 독립적입니다.

`endptr`이 `NULL`이면, 전체 문자열을 변환합니다. 문자열이 부동 소수점 숫자의 유효한 표현이 아니면 `ValueError`를 발생시키고 `-1.0`을 반환합니다.

`endptr`이 `NULL`이 아니면, 가능한 한 많은 문자열을 변환하고 `*endptr`이 변환되지 않은 첫 번째 문자를 가리키도록 설정합니다. 문자열의 초기 세그먼트가 부동 소수점 숫자의 유효한 표현이 아니면, `*endptr`이 문자열의 시작을 가리키도록 설정하고, `ValueError`를 발생시키고 `-1.0`을 반환합니다.

*s*가 `float`에 저장하기에 너무 큰 값을 나타낼 때 (예를 들어, 여러 플랫폼에서 `"1e500"`가 그런 문자열입니다), `overflow_exception`가 `NULL`이면 (적절한 부호와 함께) `Py_HUGE_VAL`을 반환하고, 어떤 예외도 설정하지 않습니다. 그렇지 않으면, `overflow_exception`은 파이썬 예외 객체를 가리켜야 합니다; 그 예외를 발생시키고 `-1.0`을 반환합니다. 두 경우 모두, 변환된 값 다음의 첫 번째 문자를 가리키도록 `*endptr`을 설정합니다.

변환 중 다른 에러가 발생하면 (예를 들어 메모리 부족 에러), 적절한 파이썬 예외를 설정하고 `-1.0`을 반환합니다.

버전 3.1에 추가.

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

제공된 *format_code*, *precision* 및 *flags*를 사용하여 `double val`을 문자열로 변환합니다.

*format_code*는 `'e'`, `'E'`, `'f'`, `'F'`, `'g'`, `'G'` 또는 `'r'` 중 하나여야 합니다. `'r'`의 경우, 제공된 *precision*은 0이어야 하며 무시됩니다. `'r'` 포맷 코드는 표준 `repr()` 형식을 지정합니다.

*flags*는 `Py_DTSF_SIGN`, `Py_DTSF_ADD_DOT_0` 또는 `Py_DTSF_ALT` 값을 0개 이상 함께 or 할 수 있습니다:

- `Py_DTSF_SIGN`은 *val*가 음수가 아닐 때도 항상 반환된 문자열 앞에 부호 문자가 오는 것을 뜻합니다.
- `Py_DTSF_ADD_DOT_0`은 반환된 문자열이 정수처럼 보이지 않도록 하는 것을 뜻합니다.
- `Py_DTSF_ALT`는 “대체” 포맷팅 규칙을 적용하는 것을 뜻합니다. 자세한 내용은 `PyOS_snprintf()` `'#'` 지정자에 대한 설명서를 참조하십시오.

*ptype*이 `NULL`이 아니면, 포인터가 가리키는 값은 `Py_DTST_FINITE`, `Py_DTST_INFINITE` 또는 `Py_DTST_NAN` 중 하나로 설정되어, *val*가 각각 유한 수, 무한 수 또는 NaN임을 나타냅니다.

반환 값은 변환된 문자열이 있는 *buffer*에 대한 포인터이거나, 변환에 실패하면 `NULL`입니다. 호출자는 `PyMem_Free()`를 호출하여 반환된 문자열을 해제해야 합니다.

버전 3.1에 추가.

int **PyOS_stricmp** (const char *s1, const char *s2)

대소 문자 구분 없는 문자열 비교. 이 함수는 대소 문자를 무시한다는 점만 제외하면 strcmp()와 거의 같게 작동합니다.

int **PyOS_strncmp** (const char *s1, const char *s2, Py_ssize_t size)

대소 문자 구분 없는 문자열 비교. 이 함수는 대소 문자를 무시한다는 점만 제외하면 strncmp()와 거의 같게 작동합니다.

6.8 리플렉션

*PyObject** **PyEval_GetBuiltins** ()

Return value: Borrowed reference. 현재 실행 프레임이나 현재 실행 중인 프레임이 없으면 스레드 상태의 인터프리터의 builtins의 딕셔너리를 반환합니다.

*PyObject** **PyEval_GetLocals** ()

Return value: Borrowed reference. 현재 실행 프레임의 지역 변수 딕셔너리를 반환하거나, 현재 실행 중인 프레임이 없으면 NULL을 반환합니다.

*PyObject** **PyEval_GetGlobals** ()

Return value: Borrowed reference. 현재 실행 프레임의 전역 변수 딕셔너리를 반환하거나, 현재 실행 중인 프레임이 없으면 NULL을 반환합니다.

*PyFrameObject** **PyEval_GetFrame** ()

Return value: Borrowed reference. 현재의 스레드 상태의 프레임을 반환합니다. 현재 실행 중인 프레임이 없으면 NULL입니다.

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

frame이 현재 실행 중인 줄 번호를 반환합니다.

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

func가 함수, 클래스 또는 인스턴스 객체면 func의 이름을 반환하고, 그렇지 않으면 func의 형의 이름을 반환합니다.

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

func의 형에 따라 설명 문자열을 반환합니다. 반환 값에는 함수 및 메서드의 “()”, “constructor”, “instance” 및 “object”가 포함됩니다. *PyEval_GetFuncName*()의 결과와 이어붙이면 func의 설명이 됩니다.

6.9 코덱 등록소와 지원 함수

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

새로운 코덱 검색 함수를 등록합니다.

부작용으로, 아직 로드되지 않았다면, encodings 패키지를 로드하여 항상 검색 함수 목록의 첫 번째 항목이 되도록 합니다.

int **PyCodec_KnownEncoding** (const char *encoding)

지정된 encoding에 대해 등록된 코덱이 있는지에 따라 1이나 0을 반환합니다. 이 함수는 항상 성공합니다.

*PyObject** **PyCodec_Encode** (*PyObject* *object, const char *encoding, const char *errors)

Return value: New reference. 일반 코덱 기반 인코딩 API.

object는 errors로 정의된 예외 처리 방법을 사용하여 지정된 encoding에 대해 발견된 인코더 함수로 전달됩니다. 코덱에 정의된 기본 방법을 사용하기 위해 errors가 NULL일 수 있습니다. 인코더를 찾을 수 없으면 LookupError를 발생시킵니다.

*PyObject** **PyCodec_Decode** (*PyObject* *object, const char *encoding, const char *errors)

Return value: New reference. 일반 코덱 기반 디코딩 API.

object는 errors로 정의된 에러 처리 방법을 사용하여 지정된 encoding에 대해 발견된 디코더 함수로 전달됩니다. 코덱에 정의된 기본 방법을 사용하기 위해 errors가 NULL 일 수 있습니다. 인코더를 찾을 수 없으면 LookupError를 발생시킵니다.

6.9.1 코덱 조회 API

다음 함수에서, encoding 문자열은 모두 소문자로 변환되어 조회되므로, 이 메커니즘을 통한 인코딩 조회는 대소문자를 구분하지 않게 됩니다. 코덱이 없으면, KeyError가 설정되고 NULL이 반환됩니다.

*PyObject** **PyCodec_Encoder** (const char *encoding)

Return value: New reference. 주어진 encoding에 대한 인코더 함수를 가져옵니다.

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

Return value: New reference. 주어진 encoding에 대한 디코더 함수를 가져옵니다.

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

Return value: New reference. 지정된 encoding에 대한 IncrementalEncoder 객체를 가져옵니다.

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

Return value: New reference. 지정된 encoding에 대한 IncrementalDecoder 객체를 가져옵니다.

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

Return value: New reference. 지정된 encoding에 대한 StreamReader 팩토리 함수를 가져옵니다.

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

Return value: New reference. 지정된 encoding에 대한 StreamWriter 팩토리 함수를 가져옵니다.

6.9.2 유니코드 인코딩 에러 처리기용 등록소 API

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

지정된 name 으로 에러 처리 콜백 함수 error를 등록합니다. 코덱이 인코딩할 수 없는 문자/디코딩할 수 없는 바이트열을 발견하고, 인코드/디코드 함수를 호출할 때 name이 error 매개 변수로 지정되었을 때 이 콜백 함수를 호출합니다.

콜백은 하나의 인자로 UnicodeEncodeError, UnicodeDecodeError 또는 UnicodeTranslateError의 인스턴스를 받아들이는데, 문제가 되는 문자나 바이트의 시퀀스와 이들의 원본 문자열에서의 오프셋에 대한 정보를 담고 있습니다 (이 정보를 추출하는 함수는 유니코드 예외 객체를 참조하세요). 콜백은 주어진 예외를 발생시키거나, 문제가 있는 시퀀스의 대체와 원래 문자열에서 인코딩/디코딩을 다시 시작해야 하는 오프셋을 제공하는 정수를 포함하는 두 항목 튜플을 반환해야 합니다.

성공하면 0을, 에러면 -1을 반환합니다.

*PyObject** **PyCodec_LookupError** (const char *name)

Return value: New reference. name 으로 등록된 에러 처리 콜백 함수를 찾습니다. 특수한 경우로 NULL이 전달될 수 있는데, 이때는 “strict”에 대한 에러 처리 콜백이 반환됩니다.

*PyObject** **PyCodec_StrictErrors** (*PyObject* *exc)

Return value: Always NULL. exc를 예외로 발생시킵니다.

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

Return value: New reference. 잘못된 입력을 건너뛰고, 유니코드 에러를 무시합니다.

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

Return value: New reference. 유니코드 인코딩 에러를 ? 나 U+FFFD로 치환합니다.

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

Return value: New reference. 유니코드 인코딩 에러를 XML 문자 참조로 치환합니다.

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

Return value: New reference. 유니코드 인코딩 에러를 백 슬래시 이스케이프(\x, \u 및 \U)로 치환합니다.

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

Return value: New reference. 유니코드 인코딩 에러를 \N{...} 이스케이프로 치환합니다.

버전 3.5에 추가.

추상 객체 계층

이 장의 함수는 객체의 형과 무관하게, 혹은 광범위한 종류의 객체 형의 (예를 들어, 모든 숫자 형 또는 모든 시퀀스 형) 파이썬 객체와 상호 작용합니다. 적용되지 않는 객체 형에 사용되면, 파이썬 예외가 발생합니다.

`PyList_New()`로 만들었지만, 항목이 아직 NULL이 아닌 값으로 설정되지 않은 리스트 객체와 같이, 제대로 초기화되지 않은 객체에 대해서는 이 함수를 사용할 수 없습니다.

7.1 객체 프로토콜

*PyObject** **Py_NotImplemented**

지정된 형 조합에 대해 연산이 구현되지 않았음을 알리는 데 사용되는 NotImplemented 싱글톤.

Py_RETURN_NOTIMPLEMENTED

C 함수 내에서 *Py_NotImplemented* 반환을 올바르게 처리합니다 (즉, NotImplemented의 참조 횟수를 증가시키고 반환합니다).

int PyObject_Print (*PyObject* *o, FILE *fp, int flags)

파일 fp에 객체 o를 인쇄합니다. 예러 시 -1을 반환합니다. flags 인자는 특정 인쇄 옵션을 활성화하는 데 사용됩니다. 현재 지원되는 유일한 옵션은 Py_PRINT_RAW입니다; 주어진다면, repr() 대신 객체의 str()이 기록됩니다.

int PyObject_HasAttr (*PyObject* *o, *PyObject* *attr_name)

o에 attr_name 어트리뷰트가 있으면 1을, 그렇지 않으면 0을 반환합니다. 이것은 파이썬 표현식 hasattr(o, attr_name)과 동등합니다. 이 함수는 항상 성공합니다.

__getattr__() 과 __getattribute__() 메서드를 호출하는 동안 발생하는 예외는 억제됨에 유의하십시오. 예러 보고를 얻으려면 대신 *PyObject_GetAttr()* 을 사용하십시오.

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

o에 attr_name 어트리뷰트가 있으면 1을, 그렇지 않으면 0을 반환합니다. 이것은 파이썬 표현식 hasattr(o, attr_name)과 동등합니다. 이 함수는 항상 성공합니다.

__getattr__() 과 __getattribute__() 메서드를 호출하고 임시 문자열 객체를 만드는 중에 발생하는 예외는 억제됨에 유의하십시오. 예러 보고를 얻으려면 대신 *PyObject_GetAttrString()* 을 사용하십시오.

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

Return value: New reference. 객체 o에서 attr_name이라는 이름의 어트리뷰트를 가져옵니다. 성공하면 어트리뷰트 값을, 실패하면 NULL을 반환합니다. 이것은 파이썬 표현식 o.attr_name과 동등합니다.

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

Return value: New reference. 객체 o에서 attr_name이라는 이름의 어트리뷰트를 가져옵니다. 성공하면 어트리뷰트 값을, 실패하면 NULL을 반환합니다. 이것은 파이썬 표현식 o.attr_name과 동등합니다.

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

Return value: New reference. 형 객체의 tp_getattro 슬롯에 배치되는 일반 어트리뷰트 게터 (getter) 함수. 객체의 (있다면) __dict__에 있는 어트리뷰트뿐만 아니라 객체의 MRO에 있는 클래스의 디렉터리에서 있는 디스크립터를 찾습니다. descriptors에 요약된 것처럼, 데이터 디스크립터는 인스턴스 어트리뷰트보다 우선하지만, 비 데이터 디스크립터는 그렇지 않습니다. 그렇지 않으면, AttributeError가 발생합니다.

int **PyObject_SetAttr** (*PyObject* *o, *PyObject* *attr_name, *PyObject* *v)

객체 o에 대해, attr_name이라는 이름의 어트리뷰트 값을 v 값으로 설정합니다. 실패 시 예외를 발생시키고 -1을 반환합니다. 성공하면 0을 반환합니다. 이것은 파이썬 문장 o.attr_name = v와 동등합니다.

v가 NULL이면, 어트리뷰트가 삭제되지만, 이 기능은 폐지되었고 *PyObject_DelAttr()*로 대체되었습니다.

int **PyObject_SetAttrString** (*PyObject* *o, const char *attr_name, *PyObject* *v)

객체 o에 대해, attr_name이라는 이름의 어트리뷰트 값을 v 값으로 설정합니다. 실패 시 예외를 발생시키고 -1을 반환합니다. 성공하면 0을 반환합니다. 이것은 파이썬 문장 o.attr_name = v와 동등합니다.

v가 NULL이면, 어트리뷰트가 삭제되지만, 이 기능은 폐지되었고 *PyObject_DelAttrString()*으로 대체되었습니다.

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

형 객체의 tp_setattro 슬롯에 배치되는 일반 어트리뷰트 세터 (setter)와 딜리터 (deleter) 함수. 객체의 MRO에 있는 클래스의 디렉터리에서 데이터 디스크립터를 찾고, 발견되면 인스턴스 디렉터리에서 있는 어트리뷰트를 설정하거나 삭제하는 것보다 우선합니다. 그렇지 않으면, 객체의 (있다면) __dict__에서 어트리뷰트가 설정되거나 삭제됩니다. 성공하면 0이 반환되고, 그렇지 않으면 AttributeError가 발생하고 -1이 반환됩니다.

int **PyObject_DelAttr** (*PyObject* *o, *PyObject* *attr_name)

객체 o에 대해, attr_name이라는 이름의 어트리뷰트를 삭제합니다. 실패 시 -1을 반환합니다. 이것은 파이썬 문장 del o.attr_name과 동등합니다.

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

객체 o에 대해, attr_name이라는 이름의 어트리뷰트를 삭제합니다. 실패 시 -1을 반환합니다. 이것은 파이썬 문장 del o.attr_name과 동등합니다.

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

Return value: New reference. __dict__ 디스크립터의 게터 (getter)를 위한 일반적인 구현. 필요하다면 디렉터리를 만듭니다.

버전 3.3에 추가.

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

__dict__ 디스크립터의 세터 (setter)를 위한 일반적인 구현. 이 구현은 디렉터리 삭제를 허락하지 않습니다.

버전 3.3에 추가.

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

Return value: New reference. opid에 의해 지정된 연산을 사용하여 o1과 o2의 값을 비교합니다. opid는 Py_LT, Py_LE, Py_EQ, Py_NE, Py_GT 또는 Py_GE 중 하나여야 하고 각각 <, <=, ==, !=, > 또는 >=에 해당합니다. 이는 파이썬 표현식 o1 op o2와 동등합니다. 여기서 op는 opid에 해당하는 연산자입니다. 성공 시 비교 값을, 실패 시 NULL을 반환합니다.

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

opid에 의해 지정된 연산을 사용하여 o1과 o2의 값을 비교합니다. opid는 Py_LT, Py_LE, Py_EQ, Py_NE, Py_GT 또는 Py_GE 중 하나여야 하고 각각 <, <=, ==, !=, > 또는 >=에 해당합니다. 예러 시 -1을, 결과가 거짓이면 0을, 그렇지 않으면 1을 반환합니다. 이는 파이썬 표현식 o1 op o2와 동등합니다. 여기서 op는 opid에 해당하는 연산자입니다.

참고: o1과 o2가 같은 객체이면, PyObject_RichCompareBool()은 항상 Py_EQ의 경우는 1을, Py_NE의 경우는 0을 반환합니다.

PyObject* PyObject_Repr (PyObject *o)

Return value: New reference. 객체 o의 문자열 표현을 계산합니다. 성공하면 문자열 표현을, 실패하면 NULL을 반환합니다. 이것은 파이썬 표현식 repr(o)와 동등합니다. repr() 내장 함수에 의해 호출됩니다.

버전 3.4에서 변경: 이 함수에는 이제 디버그 어서션이 포함되어 있어 활성 예외를 조용히 버리지 않도록 합니다.

PyObject* PyObject_ASCII (PyObject *o)

Return value: New reference. PyObject_Repr()처럼, 객체 o의 문자열 표현을 계산하지만, \x, \u 또는 \U 이스케이프를 사용하여 PyObject_Repr()이 반환한 문자열에서 비 ASCII 문자를 이스케이프합니다. 이것은 파이썬 2에서 PyObject_Repr()에 의해 반환된 것과 유사한 문자열을 생성합니다. ascii() 내장 함수에 의해 호출됩니다.

PyObject* PyObject_Str (PyObject *o)

Return value: New reference. 객체 o의 문자열 표현을 계산합니다. 성공 시 문자열 표현을, 실패 시 NULL을 반환합니다. 이것은 파이썬 표현식 str(o)와 동등합니다. str() 내장 함수에 의해, 따라서 print() 함수에 의해서도 호출됩니다.

버전 3.4에서 변경: 이 함수에는 이제 디버그 어서션이 포함되어 있어 활성 예외를 조용히 버리지 않도록 합니다.

PyObject* PyObject_Bytes (PyObject *o)

Return value: New reference. 객체 o의 바이트열 표현을 계산합니다. 실패하면 NULL을, 성공하면 바이트열 객체를 반환합니다. 이는 o가 정수가 아닐 때 파이썬 표현식 bytes(o)와 동등합니다. bytes(o)와 달리, o가 정수이면 0으로 초기화된 바이트열 객체 대신 TypeError가 발생합니다.

int PyObject_IsSubclass (PyObject *derived, PyObject *cls)

클래스 derived가 클래스 cls와 동일하거나 cls에서 파생되었으면 1을 반환하고, 그렇지 않으면 0을 반환합니다. 예러가 발생하면 -1을 반환합니다.

cls가 튜플이면, cls의 모든 항목에 대해 검사가 수행됩니다. 적어도 하나의 검사에서 1을 반환하면 결과는 1이 되고, 그렇지 않으면 0이 됩니다.

cls에 __subclasscheck__() 메서드가 있으면, PEP 3119에 설명된 대로 서브 클래스 상태를 판별하기 위해 호출됩니다. 그렇지 않으면, derived가 직접 또는 간접 서브 클래스일 때 cls의 서브 클래스입니다, 즉 cls.__mro__에 포함되어 있습니다.

일반적으로 클래스 객체(즉 type이나 파생 클래스의 인스턴스)만 클래스로 간주합니다. 그러나, 객체는 __bases__ 어트리뷰트(베이스 클래스의 튜플이어야 합니다)를 가짐으로써 이를 재정의할 수 있습니다.

int PyObject_IsInstance (PyObject *inst, PyObject *cls)

inst가 cls 클래스나 cls의 서브 클래스의 인스턴스이면 1을 반환하고, 그렇지 않으면 0을 반환합니다. 예러가 발생하면 -1을 반환하고 예외를 설정합니다.

cls가 튜플이면, cls의 모든 항목에 대해 검사가 수행됩니다. 적어도 하나의 검사에서 1을 반환하면 결과는 1이 되고, 그렇지 않으면 0이 됩니다.

cls에 __instancecheck__() 메서드가 있으면, PEP 3119에 설명된 대로 서브 클래스 상태를 판별하기 위해 호출됩니다. 그렇지 않으면, inst는 해당 클래스가 cls의 서브 클래스일 때 cls의 인스턴스입니다.

인스턴스 *inst*는 `__class__` 어트리뷰트를 가짐으로써 클래스로 간주하는 것을 재정의할 수 있습니다.

객체 *cls*는 `__bases__` 어트리뷰트(베이스 클래스의 튜플이어야 합니다)를 가짐으로써, 클래스로 간주하는지와 베이스 클래스가 무엇인지를 재정의할 수 있습니다.

int PyCallable_Check (*PyObject* *o)

객체 *o*가 콜러블인지 판별합니다. 객체가 콜러블이면 1을, 그렇지 않으면 0을 반환합니다. 이 함수는 항상 성공합니다.

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

Return value: New reference. 튜플 *args*로 제공된 인자와 딕셔너리 *kwargs*로 제공된 이름있는 인자로 콜러블 파이썬 객체 *callable*을 호출합니다.

*args*는 NULL이 아니어야 합니다, 인자가 필요하지 않으면 빈 튜플을 사용하십시오. 이름있는 인자가 필요하지 않으면, *kwargs*는 NULL일 수 있습니다.

성공하면 호출 결과를 반환하고, 실패하면 예외를 발생시키고 NULL을 반환합니다.

이것은 파이썬 표현식 `callable(*args, **kwargs)`와 동등합니다.

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

Return value: New reference. 튜플 *args*로 제공된 인자로 콜러블 파이썬 객체 *callable*을 호출합니다. 인자가 필요하지 않으면, *args*는 NULL일 수 있습니다.

성공하면 호출 결과를 반환하고, 실패하면 예외를 발생시키고 NULL을 반환합니다.

이것은 파이썬 표현식 `callable(*args)`와 동등합니다.

PyObject* PyObject_CallFunction (*PyObject* *callable, const char *format, ...)

Return value: New reference. 가변 개수의 C 인자로 콜러블 파이썬 객체 *callable*을 호출합니다. C 인자는 `Py_BuildValue()` 스타일 포맷 문자열을 사용하여 기술됩니다. *format*은 NULL일 수 있으며, 인자가 제공되지 않았음을 나타냅니다.

성공하면 호출 결과를 반환하고, 실패하면 예외를 발생시키고 NULL을 반환합니다.

이것은 파이썬 표현식 `callable(*args)`와 동등합니다.

Note that if you only pass *PyObject* *args, `PyObject_CallFunctionObjArgs()` is a faster alternative.

버전 3.4에서 변경: *format*의 형이 `char *`에서 변경되었습니다.

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

Return value: New reference. 가변 개수의 C 인자로 객체 *obj*의 *name*이라는 메서드를 호출합니다. C 인자는 튜플을 생성해야 하는 `Py_BuildValue()` 포맷 문자열로 기술됩니다.

*format*은 NULL일 수 있으며, 인자가 제공되지 않았음을 나타냅니다.

성공하면 호출 결과를 반환하고, 실패하면 예외를 발생시키고 NULL을 반환합니다.

이것은 파이썬 표현식 `obj.name(arg1, arg2, ...)`와 동등합니다.

Note that if you only pass *PyObject* *args, `PyObject_CallMethodObjArgs()` is a faster alternative.

버전 3.4에서 변경: *name*과 *format*의 형이 `char *`에서 변경되었습니다.

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.

성공하면 호출 결과를 반환하고, 실패하면 예외를 발생시키고 NULL을 반환합니다.

이것은 파이썬 표현식 `callable(arg1, arg2, ...)`와 동등합니다.

PyObject* PyObject_CallMethodObjArgs (*PyObject* *obj, *PyObject* *name, ...)

Return value: New reference. Calls 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.

성공하면 호출 결과를 반환하고, 실패하면 예외를 발생시키고 NULL을 반환합니다.

*PyObject** **_PyObject_Vectorcall** (*PyObject** callable, *PyObject** const *args, size_t nargsf, *PyObject** kw-names)

가능하면 벡터콜을 사용하여, 콜러블 파이썬 객체 *callable*을 호출합니다.

*args*는 위치 인자가 있는 C 배열입니다.

*nargsf*는 위치 인자의 수와 선택적인 플래그 PY_VECTORCALL_ARGUMENTS_OFFSET입니다 (아래를 참조하십시오). 실제 인자의 개수를 얻으려면, *PyVectorcall_NARGS(nargsf)*를 사용하십시오.

*kw-names*는 NULL(키워드 인자 없음)이나 키워드 이름의 튜플일 수 있습니다. 후자의 경우, 키워드 인자의 값은 위치 인자 다음에 *args*에 저장됩니다. 키워드 인자의 수는 *nargsf*에 영향을 미치지 않습니다.

*kw-names*는 str 형(서브 클래스는 아닙니다)의 객체만 포함해야 하며, 모든 키는 고유해야 합니다.

성공하면 호출 결과를 반환하고, 실패하면 예외를 발생시키고 NULL을 반환합니다.

*callable*이 지원한다면 벡터콜 프로토콜을 사용합니다; 그렇지 않으면, 인자는 *tp_call*을 사용하도록 변환됩니다.

참고: 이 함수는 잠정적이며 파이썬 3.9에서 다른 이름으로 변경되고 어쩌면 의미도 변경되어 공개될 예정입니다. 이 함수를 사용한다면, 파이썬 3.9를 위해 코드를 변경할 준비를 하십시오.

버전 3.8에 추가.

PY_VECTORCALL_ARGUMENTS_OFFSET

벡터콜 *nargsf* 인자에 설정되면, 피호출자는 일시적으로 *args*[-1]을 변경할 수 있습니다. 즉, *args*는 할당된 벡터에서 인자 1(0이 아닙니다)을 가리킵니다. 피호출자는 반환하기 전에 *args*[-1] 값을 복원해야 합니다.

(추가 할당 없이) 저렴하게 할 수 있을 때마다, 호출자는 PY_VECTORCALL_ARGUMENTS_OFFSET을 사용하는 것이 좋습니다. 이렇게 하면 연결된 메서드와 같은 콜러블 항목이 선행 호출(*self* 인자를 앞에 붙이는 것을 포함합니다)을 저렴하게 만들 수 있습니다.

버전 3.8에 추가.

Py_ssize_t **PyVectorcall_NARGS** (size_t nargsf)

주어진 벡터콜 *nargsf* 인자에서 실제 인자 수를 반환합니다. 현재 *nargsf* & ~PY_VECTORCALL_ARGUMENTS_OFFSET과 동등합니다.

버전 3.8에 추가.

*PyObject** **_PyObject_FastCallDict** (*PyObject** callable, *PyObject** const *args, size_t nargsf, *PyObject** kwdict)

키워드 인자가 *kwdict* 딕셔너리로 전달된다는 점을 제외하고는 *_PyObject_Vectorcall()*과 같습니다. 키워드 인자가 없으면 *kwdict*는 NULL일 수 있습니다.

벡터콜을 지원하는 콜러블의 경우, 인자는 내부적으로 벡터콜 규칙으로 변환됩니다. 따라서, 이 함수는 *_PyObject_Vectorcall()*에 비해 약간의 오버헤드를 추가합니다. 호출자에게 이미 사용할 준비가 된 딕셔너리가 있을 때만 사용해야 합니다.

참고: 이 함수는 잠정적이며 파이썬 3.9에서 다른 이름으로 변경되고 어쩌면 의미도 변경되어 공개될 예정입니다. 이 함수를 사용한다면, 파이썬 3.9를 위해 코드를 변경할 준비를 하십시오.

버전 3.8에 추가.

Py_hash_t PyObject_Hash (*PyObject *o*)

객체 *o*의 해시값을 계산하고 반환합니다. 실패하면 -1을 반환합니다. 이것은 파이썬 표현식 `hash(o)`와 동등합니다.

버전 3.2에서 변경: 반환형은 이제 `Py_hash_t`입니다. 이것은 `Py_ssize_t`와 같은 크기의 부호 있는 정수입니다.

Py_hash_t PyObject_HashNotImplemented (*PyObject *o*)

`type(o)`가 해시 가능하지 않음을 나타내는 `TypeError`를 설정하고 -1을 반환합니다. 이 함수는 `tp_hash` 슬롯에 저장될 때 특수한 처방을 받아서, 인터프리터에 형이 해시 가능하지 않음을 명시적으로 알립니다.

int PyObject_IsTrue (*PyObject *o*)

객체 *o*를 참으로 간주하면 1을, 그렇지 않으면 0을 반환합니다. 이것은 파이썬 표현식 `not not o`와 동등합니다. 실패하면 -1을 반환합니다.

int PyObject_Not (*PyObject *o*)

객체 *o*를 참으로 간주하면 0을, 그렇지 않으면 1을 반환합니다. 이것은 파이썬 표현식 `not o`와 동등합니다. 실패하면 -1을 반환합니다.

PyObject* PyObject_Type (*PyObject *o*)

Return value: New reference. *o*가 NULL이 아니면, 객체 *o*의 객체 형에 해당하는 형 객체를 반환합니다. 실패하면 `SystemError`를 발생시키고 NULL을 반환합니다. 이것은 파이썬 표현식 `type(o)`와 동등합니다. 이 함수는 반환 값의 참조 횟수를 증가시킵니다. 증가한 참조 횟수가 필요할 때를 제외하고, `PyTypeObject*` 형의 포인터를 반환하는 공통 표현식 `o->ob_type` 대신 이 함수를 사용할 이유가 없습니다.

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

객체 *o*가 *type* 형이거나 *type*의 서브 형이면 참을 반환합니다. 두 매개 변수 모두 NULL이 아니어야 합니다.

Py_ssize_t PyObject_Size (*PyObject *o*)

Py_ssize_t PyObject_Length (*PyObject *o*)

객체 *o*의 길이를 반환합니다. 객체 *o*가 시퀀스와 매핑 프로토콜을 제공하면, 시퀀스 길이가 반환됩니다. 예러가 발생하면 -1이 반환됩니다. 이것은 파이썬 표현식 `len(o)`와 동등합니다.

Py_ssize_t PyObject_LengthHint (*PyObject *o, Py_ssize_t default*)

o 객체의 추정된 길이를 반환합니다. 먼저 실제 길이를 반환하려고 시도한 다음, `__length_hint__()`를 사용하여 추정값을 반환하고, 마지막으로 기본값을 반환합니다. 예러 시 -1을 반환합니다. 이것은 파이썬 표현식 `operator.length_hint(o, default)`와 동등합니다.

버전 3.4에 추가.

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

Return value: New reference. 객체 *key*에 해당하는 *o*의 요소를 반환하거나 실패 시 NULL을 반환합니다. 이것은 파이썬 표현식 `o[key]`와 동등합니다.

int PyObject_SetItem (*PyObject *o, PyObject *key, PyObject *v*)

객체 *key*를 값 *v*에 매핑합니다. 실패 시 예외를 발생시키고 -1을 반환합니다; 성공하면 0을 반환합니다. 이것은 파이썬 문장 `o[key] = v`와 동등합니다. 이 함수는 *v*에 대한 참조를 훔치지 않습니다.

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

객체 *o*에서 객체 *key*에 대한 매핑을 제거합니다. 실패하면 -1을 반환합니다. 이것은 파이썬 문장 `del o[key]`와 동등합니다.

PyObject* PyObject_Dir (*PyObject *o*)

Return value: New reference. 이것은 파이썬 표현식 `dir(o)`와 동등하며, 객체 인자에 적합한 문자열의 (비어있을 수 있는) 리스트를 반환하거나, 예러가 있으면 NULL을 반환합니다. 인자가 NULL이면, 파이썬 `dir()`과 비슷하며, 현재 지역(locals)의 이름들을 반환합니다; 이 경우, 실행 프레임이 활성화되어 있지 않으면 NULL이 반환되지만 `PyErr_Occurred()`는 거짓을 반환합니다.

*PyObject** **PyObject_GetIter** (*PyObject* *o)

Return value: New reference. 이것은 파이썬 표현식 `iter(o)`와 동등합니다. 객체 인자에 대한 새로운 이터레이터를 반환하거나, 객체가 이미 이터레이터이면 객체 자체를 반환합니다. 객체를 이터레이트 할 수 없으면 `TypeError`를 발생시키고 `NULL`을 반환합니다.

7.2 숫자 프로토콜

int **PyNumber_Check** (*PyObject* *o)

객체 *o*가 숫자 프로토콜을 제공하면 1을 반환하고, 그렇지 않으면 거짓을 반환합니다. 이 함수는 항상 성공합니다.

버전 3.8에서 변경: *o*가 인덱스 정수면 1을 반환합니다.

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

Return value: New reference. *o1*과 *o2*를 더한 결과나, 실패 시 `NULL`을 반환합니다. 이것은 파이썬 표현식 `o1 + o2`와 동등합니다.

*PyObject** **PyNumber_Subtract** (*PyObject* *o1, *PyObject* *o2)

Return value: New reference. *o1*에서 *o2*를 뺀 결과나, 실패 시 `NULL`을 반환합니다. 이것은 파이썬 표현식 `o1 - o2`와 동등합니다.

*PyObject** **PyNumber_Multiply** (*PyObject* *o1, *PyObject* *o2)

Return value: New reference. *o1*과 *o2*를 곱한 결과나, 실패 시 `NULL`을 반환합니다. 이것은 파이썬 표현식 `o1 * o2`와 동등합니다.

*PyObject** **PyNumber_MatrixMultiply** (*PyObject* *o1, *PyObject* *o2)

Return value: New reference. *o1*과 *o2*를 행렬 곱셈한 결과나, 실패 시 `NULL`을 반환합니다. 이것은 파이썬 표현식 `o1 @ o2`와 동등합니다.

버전 3.5에 추가.

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

Return value: New reference. *o1*을 *o2*로 나눈 결과나, 실패 시 `NULL`을 반환합니다. 이것은 “고전적인” 정수 나눗셈과 동등합니다.

*PyObject** **PyNumber_TrueDivide** (*PyObject* *o1, *PyObject* *o2)

Return value: New reference. *o1*을 *o2*로 나눈 수학적 값의 적절한 근삿값이나, 실패 시 `NULL`을 반환합니다. 반환 값은 “근사치” 인데, 이진 부동 소수점 수가 근사치이기 때문입니다; 이진수로 모든 실수를 표현할 수는 없습니다. 이 함수는 두 개의 정수를 전달할 때 부동 소수점 수를 반환할 수 있습니다.

*PyObject** **PyNumber_Remainder** (*PyObject* *o1, *PyObject* *o2)

Return value: New reference. *o1*을 *o2*로 나눈 나머지가, 실패 시 `NULL`을 반환합니다. 이것은 파이썬 표현식 `o1 % o2`와 동등합니다.

*PyObject** **PyNumber_Divmod** (*PyObject* *o1, *PyObject* *o2)

Return value: New reference. 내장 함수 `divmod()`를 참조하십시오. 실패하면 `NULL`을 반환합니다. 이것은 파이썬 표현식 `divmod(o1, o2)`와 동등합니다.

*PyObject** **PyNumber_Power** (*PyObject* *o1, *PyObject* *o2, *PyObject* *o3)

Return value: New reference. 내장 함수 `pow()`를 참조하십시오. 실패하면 `NULL`을 반환합니다. 이것은 파이썬 표현식 `pow(o1, o2, o3)`와 동등합니다, 여기서 *o3*는 선택적입니다. *o3*를 무시하려면, 그 자리에 `Py_None`을 전달하십시오 (*o3*에 `NULL`을 전달하면 잘못된 메모리 액세스가 발생합니다).

*PyObject** **PyNumber_Negative** (*PyObject* *o)

Return value: New reference. 성공 시 *o*의 음의 값(negation)을, 실패 시 `NULL`을 반환합니다. 이것은 파이썬 표현식 `-o`와 동등합니다.

*PyObject** **PyNumber_Positive** (*PyObject* *o)

Return value: New reference. 성공 시 o를, 실패 시 NULL을 반환합니다. 이것은 파이썬 표현식 +o와 동등합니다.

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

Return value: New reference. o의 절댓값이나, 실패 시 NULL을 반환합니다. 이것은 파이썬 표현식 abs(o)와 동등합니다.

*PyObject** **PyNumber_Invert** (*PyObject* *o)

Return value: New reference. 성공 시 o의 비트 반전 (bitwise negation)을, 실패 시 NULL을 반환합니다. 이것은 파이썬 표현식 ~o와 동등합니다.

*PyObject** **PyNumber_Lshift** (*PyObject* *o1, *PyObject* *o2)

Return value: New reference. 성공 시 o1을 o2만큼 왼쪽으로 시프트 한 결과를, 실패 시 NULL을 반환합니다. 이것은 파이썬 표현식 o1 << o2와 동등합니다.

*PyObject** **PyNumber_Rshift** (*PyObject* *o1, *PyObject* *o2)

Return value: New reference. 성공 시 o1을 o2만큼 오른쪽으로 시프트 한 결과를, 실패 시 NULL을 반환합니다. 이것은 파이썬 표현식 o1 >> o2와 동등합니다.

*PyObject** **PyNumber_And** (*PyObject* *o1, *PyObject* *o2)

Return value: New reference. 성공 시 o1과 o2의 “비트별 논리곱 (bitwise and)”을, 실패 시 NULL을 반환합니다. 이것은 파이썬 표현식 o1 & o2와 동등합니다.

*PyObject** **PyNumber_Xor** (*PyObject* *o1, *PyObject* *o2)

Return value: New reference. 성공 시 o1과 o2의 “비트별 배타적 논리합 (bitwise exclusive or)”을, 실패 시 NULL을 반환합니다. 이것은 파이썬 표현식 o1 ^ o2와 동등합니다.

*PyObject** **PyNumber_Or** (*PyObject* *o1, *PyObject* *o2)

Return value: New reference. 성공 시 o1과 o2의 “비트별 논리합 (bitwise or)”을, 실패 시 NULL을 반환합니다. 이것은 파이썬 표현식 o1 | o2와 동등합니다.

*PyObject** **PyNumber_InPlaceAdd** (*PyObject* *o1, *PyObject* *o2)

Return value: New reference. o1과 o2를 더한 결과나, 실패 시 NULL을 반환합니다. 이 연산은 o1이 지원하면 제자리에서 (in-place) 수행됩니다. 이것은 파이썬 문장 o1 += o2와 동등합니다.

*PyObject** **PyNumber_InPlaceSubtract** (*PyObject* *o1, *PyObject* *o2)

Return value: New reference. o1에서 o2를 뺀 결과나, 실패 시 NULL을 반환합니다. 이 연산은 o1이 지원하면 제자리에서 (in-place) 수행됩니다. 이것은 파이썬 문장 o1 -= o2와 동등합니다.

*PyObject** **PyNumber_InPlaceMultiply** (*PyObject* *o1, *PyObject* *o2)

Return value: New reference. o1과 o2를 곱한 결과나, 실패 시 NULL을 반환합니다. 이 연산은 o1이 지원하면 제자리에서 (in-place) 수행됩니다. 이것은 파이썬 문장 o1 *= o2와 동등합니다.

*PyObject** **PyNumber_InPlaceMatrixMultiply** (*PyObject* *o1, *PyObject* *o2)

Return value: New reference. o1과 o2를 행렬 곱셈한 결과나, 실패 시 NULL을 반환합니다. 이 연산은 o1이 지원하면 제자리에서 (in-place) 수행됩니다. 이것은 파이썬 문장 o1 @= o2와 동등합니다.

버전 3.5에 추가.

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

Return value: New reference. o1을 o2로 나눈 수학적 플로어 (floor)나, 실패 시 NULL을 반환합니다. 이 연산은 o1이 지원하면 제자리에서 (in-place) 수행됩니다. 이것은 파이썬 문장 o1 //= o2와 동등합니다.

*PyObject** **PyNumber_InPlaceTrueDivide** (*PyObject* *o1, *PyObject* *o2)

Return value: New reference. o1을 o2로 나눈 수학적 값의 적절한 근삿값이나, 실패 시 NULL을 반환합니다. 반환 값은 “근사치” 인데, 이진 부동 소수점 수가 근사치이기 때문입니다; 이진수로 모든 실수를 표현할 수는 없습니다. 이 함수는 두 개의 정수를 전달할 때 부동 소수점 수를 반환할 수 있습니다. 이 연산은 o1이 지원하면 제자리에서 (in-place) 수행됩니다.

*PyObject** **PyNumber_InPlaceRemainder** (*PyObject* *o1, *PyObject* *o2)

Return value: New reference. o1을 o2로 나눈 나머지가, 실패 시 NULL을 반환합니다. 이 연산은 o1이 지원하면 제자리에서 (in-place) 수행됩니다. 이것은 파이썬 문장 o1 %= o2와 동등합니다.

*PyObject** **PyNumber_InPlacePower** (*PyObject* *o1, *PyObject* *o2, *PyObject* *o3)

Return value: New reference. 내장 함수 pow()를 참조하십시오. 실패하면 NULL을 반환합니다. 이 연산은 o1이 지원하면 제자리에서 (in-place) 수행됩니다. 이것은 o3가 Py_None일 때 파이썬 문장 o1 **= o2와, 그렇지 않으면 pow(o1, o2, o3)의 제자리 변형과 동등합니다. o3를 무시하려면, 그 자리에 Py_None을 전달하십시오 (o3에 NULL을 전달하면 잘못된 메모리 액세스가 발생합니다).

*PyObject** **PyNumber_InPlaceLshift** (*PyObject* *o1, *PyObject* *o2)

Return value: New reference. 성공 시 o1을 o2만큼 왼쪽으로 시프트 한 결과를, 실패 시 NULL을 반환합니다. 이 연산은 o1이 지원하면 제자리에서 (in-place) 수행됩니다. 이것은 파이썬 문장 o1 <<= o2와 동등합니다.

*PyObject** **PyNumber_InPlaceRshift** (*PyObject* *o1, *PyObject* *o2)

Return value: New reference. 성공 시 o1을 o2만큼 오른쪽으로 시프트 한 결과를, 실패 시 NULL을 반환합니다. 이 연산은 o1이 지원하면 제자리에서 (in-place) 수행됩니다. 이것은 파이썬 문장 o1 >>= o2와 동등합니다.

*PyObject** **PyNumber_InPlaceAnd** (*PyObject* *o1, *PyObject* *o2)

Return value: New reference. 성공 시 o1과 o2의 “비트별 논리곱 (bitwise and)”을, 실패 시 NULL을 반환합니다. 이 연산은 o1이 지원하면 제자리에서 (in-place) 수행됩니다. 이것은 파이썬 문장 o1 &= o2와 동등합니다.

*PyObject** **PyNumber_InPlaceXor** (*PyObject* *o1, *PyObject* *o2)

Return value: New reference. 성공 시 o1과 o2의 “비트별 배타적 논리합 (bitwise exclusive or)”을, 실패 시 NULL을 반환합니다. 이 연산은 o1이 지원하면 제자리에서 (in-place) 수행됩니다. 이것은 파이썬 문장 o1 ^= o2와 동등합니다.

*PyObject** **PyNumber_InPlaceOr** (*PyObject* *o1, *PyObject* *o2)

Return value: New reference. 성공 시 o1과 o2의 “비트별 논리합 (bitwise or)”을, 실패 시 NULL을 반환합니다. 이 연산은 o1이 지원하면 제자리에서 (in-place) 수행됩니다. 이것은 파이썬 문장 o1 |= o2와 동등합니다.

*PyObject** **PyNumber_Long** (*PyObject* *o)

Return value: New reference. 성공 시 정수 객체로 변환된 o를, 실패 시 NULL을 반환합니다. 이것은 파이썬 표현식 int(o)와 동등합니다.

*PyObject** **PyNumber_Float** (*PyObject* *o)

Return value: New reference. 성공 시 float 객체로 변환된 o를, 실패 시 NULL을 반환합니다. 이것은 파이썬 표현식 float(o)와 동등합니다.

*PyObject** **PyNumber_Index** (*PyObject* *o)

Return value: New reference. 성공 시 파이썬 int로 변환된 o를, 실패 시 NULL을 반환합니다. 실패 시 TypeError 예외가 발생합니다.

*PyObject** **PyNumber_ToBase** (*PyObject* *n, int base)

Return value: New reference. 정수 n을 진수 base를 사용해서 변환한 문자열을 반환합니다. base 인자는 2, 8, 10 또는 16중 하나여야 합니다. 진수 2, 8 또는 16의 경우, 반환된 문자열은 '0b', '0o' 또는 '0x'의 진수 표시자가 각각 앞에 붙습니다. n이 파이썬 int가 아니면, 먼저 PyNumber_Index()로 변환됩니다.

Py_ssize_t **PyNumber_AsSsize_t** (*PyObject* *o, *PyObject* *exc)

o가 정수로 해석될 수 있으면, o를 Py_ssize_t 값으로 변환하여 반환합니다. 호출이 실패하면, 예외가 발생하고 -1이 반환됩니다.

o가 파이썬 int로 변환될 수 있지만 Py_ssize_t 값으로 변환하려는 시도가 OverflowError를 발생시키면, exc 인자는 발생할 예외의 형 (일반적으로 IndexError나 OverflowError)입니다. exc가 NULL이면, 예외가 지워지고 값은 음의 정수는 PY_SSIZE_T_MIN으로, 양의 정수는 PY_SSIZE_T_MAX로 잘립니다.

int **PyIndex_Check** (*PyObject* *o)

*o*가 인덱스 정수(`tp_as_number` 구조의 `nb_index` 슬롯이 채워져 있습니다)면 1을 반환하고, 그렇지 않으면 0을 반환합니다. 이 함수는 항상 성공합니다.

7.3 시퀀스 프로토콜

int **PySequence_Check** (*PyObject* *o)

객체가 시퀀스 프로토콜을 제공하면 1을 반환하고, 그렇지 않으면 0을 반환합니다. `__getitem__()` 메서드가 있는 파이썬 클래스의 경우 `dict` 서브 클래스가 아닌 한 1을 반환하는 것에 유의하십시오. 일반적으로 어떤 형의 키를 지원하는지 판단할 수 없기 때문입니다. 이 함수는 항상 성공합니다.

Py_ssize_t **PySequence_Size** (*PyObject* *o)

Py_ssize_t **PySequence_Length** (*PyObject* *o)

성공 시 시퀀스 *o*의 객체 수를 반환하고, 실패하면 -1을 반환합니다. 이것은 파이썬 표현식 `len(o)`와 동등합니다.

*PyObject** **PySequence_Concat** (*PyObject* *o1, *PyObject* *o2)

Return value: New reference. 성공 시 *o1*와 *o2*의 이어붙이기를 반환하고, 실패하면 NULL을 반환합니다. 이것은 파이썬 표현식 `o1 + o2`와 동등합니다.

*PyObject** **PySequence_Repeat** (*PyObject* *o, *Py_ssize_t* count)

Return value: New reference. 시퀀스 객체 *o*를 *count* 번 반복한 결과를 반환하거나, 실패 시 NULL을 반환합니다. 이것은 파이썬 표현식 `o * count`와 동등합니다.

*PyObject** **PySequence_InPlaceConcat** (*PyObject* *o1, *PyObject* *o2)

Return value: New reference. 성공 시 *o1*와 *o2*의 이어붙이기를 반환하고, 실패하면 NULL을 반환합니다. 이 연산은 *o1*가 지원하면 제자리에서(*in-place*) 수행됩니다. 이것은 파이썬 표현식 `o1 += o2`와 동등합니다.

*PyObject** **PySequence_InPlaceRepeat** (*PyObject* *o, *Py_ssize_t* count)

Return value: New reference. 시퀀스 객체 *o*를 *count* 번 반복한 결과를 반환하거나, 실패 시 NULL을 반환합니다. 이 연산은 *o*가 지원하면 제자리에서(*in-place*) 수행됩니다. 이것은 파이썬 표현식 `o *= count`와 동등합니다.

*PyObject** **PySequence_GetItem** (*PyObject* *o, *Py_ssize_t* i)

Return value: New reference. *o*의 *i* 번째 요소를 반환하거나, 실패하면 NULL을 반환합니다. 이것은 파이썬 표현식 `o[i]`와 동등합니다.

*PyObject** **PySequence_GetSlice** (*PyObject* *o, *Py_ssize_t* i1, *Py_ssize_t* i2)

Return value: New reference. 시퀀스 객체 *o*의 *i1*와 *i2* 사이의 슬라이스를 반환하거나, 실패하면 NULL을 반환합니다. 이것은 파이썬 표현식 `o[i1:i2]`와 동등합니다.

int **PySequence_SetItem** (*PyObject* *o, *Py_ssize_t* i, *PyObject* *v)

객체 *v*를 *o*의 *i* 번째 요소에 대입합니다. 실패하면 예외를 발생시키고 -1을 반환합니다; 성공하면 0을 반환합니다. 이것은 파이썬 문장 `o[i] = v`와 동등합니다. 이 함수는 *v*에 대한 참조를 훔치지 않습니다.

*v*가 NULL이면, 요소가 삭제되지만, 이 기능은 `PySequence_DelItem()` 사용을 위해 폐지되었습니다.

int **PySequence_DelItem** (*PyObject* *o, *Py_ssize_t* i)

o 객체의 *i* 번째 요소를 삭제합니다. 실패하면 -1을 반환합니다. 이것은 파이썬 문장 `del o[i]`와 동등합니다.

int **PySequence_SetSlice** (*PyObject* *o, *Py_ssize_t* i1, *Py_ssize_t* i2, *PyObject* *v)

시퀀스 객체 *v*를 시퀀스 객체 *o*의 *i1*에서 *i2* 사이의 슬라이스에 대입합니다. 이것은 파이썬 문장 `o[i1:i2] = v`와 동등합니다.

int **PySequence_DelSlice** (*PyObject* *o, *Py_ssize_t* i1, *Py_ssize_t* i2)

시퀀스 객체 *o*의 *i1*에서 *i2* 사이의 슬라이스를 삭제합니다. 실패하면 -1을 반환합니다. 이것은 파이썬 문장 `del o[i1:i2]`와 동등합니다.

`Py_ssize_t PySequence_Count (PyObject *o, PyObject *value)`

*o*에 있는 *value*의 수를 반환합니다. 즉, `o[key] == value`를 만족하는 *key*의 수를 반환합니다. 실패하면 -1을 반환합니다. 이것은 파이썬 표현식 `o.count (value)`와 동등합니다.

`int PySequence_Contains (PyObject *o, PyObject *value)`

*o*에 *value*가 있는지 확인합니다. *o*의 항목 중 하나가 *value*와 같으면 1을 반환하고, 그렇지 않으면 0을 반환합니다. 에러 시 -1을 반환합니다. 이는 파이썬 표현식 `value in o`와 동등합니다.

`Py_ssize_t PySequence_Index (PyObject *o, PyObject *value)`

`o[i] == value`를 만족하는 첫 번째 인덱스 *i*를 반환합니다. 에러 시 -1을 반환합니다. 이것은 파이썬 표현식 `o.index (value)`와 동등합니다.

`PyObject* PySequence_List (PyObject *o)`

Return value: New reference. 시퀀스나 이터러블 *o*와 같은 내용을 가진 리스트 객체를 반환하거나, 실패하면 NULL을 반환합니다. 반환된 리스트는 새로운 것으로 보장됩니다. 이것은 파이썬 표현식 `list (o)`와 동등합니다.

`PyObject* PySequence_Tuple (PyObject *o)`

Return value: New reference. 시퀀스나 이터러블 *o*와 같은 내용을 가진 튜플 객체를 반환하거나, 실패하면 NULL을 반환합니다. *o*가 튜플이면, 새로운 참조가 반환되고, 그렇지 않으면 튜플이 적절한 내용으로 만들어집니다. 이것은 파이썬 표현식 `tuple (o)`와 동등합니다.

`PyObject* PySequence_Fast (PyObject *o, const char *m)`

Return value: New reference. 시퀀스나 이터러블 *o*를 다른 `PySequence_Fast*` 계열 함수에서 사용할 수 있는 객체로 반환합니다. 객체가 시퀀스나 이터러블이 아니면 *m*을 메시지 텍스트로 사용하여 `TypeError`를 발생시킵니다. 실패 시 NULL을 반환합니다.

`PySequence_Fast*` 함수는 *o*가 `PyTupleObject`나 `PyListObject`라고 가정하고 *o*의 데이터 필드에 직접 액세스하기 때문에 이렇게 이름 붙였습니다.

CPython 구현 세부 사항으로, *o*가 이미 시퀀스나 리스트면, 반환됩니다.

`Py_ssize_t PySequence_Fast_GET_SIZE (PyObject *o)`

*o*의 길이를 반환하는데, *o*가 `PySequence_Fast()`에 의해 반환되었고, *o*가 NULL이 아니라고 가정합니다. 크기는 *o*에 대해 `PySequence_Size()`를 호출하여 얻을 수도 있지만, `PySequence_Fast_GET_SIZE()`는 *o*가 리스트나 튜플이라고 가정할 수 있으므로 더 빠릅니다.

`PyObject* PySequence_Fast_GET_ITEM (PyObject *o, Py_ssize_t i)`

Return value: Borrowed reference. *o*의 *i* 번째 요소를 반환하는데, *o*가 `PySequence_Fast()`에 의해 반환되었고, *o*가 NULL이 아니며, *i*가 경계 내에 있다고 가정합니다.

`PyObject** PySequence_Fast_ITEMS (PyObject *o)`

PyObject 포인터의 하부 배열을 반환합니다. *o*가 `PySequence_Fast()`에 의해 반환되었고, *o*가 NULL이 아니라고 가정합니다.

리스트의 크기가 변경되면, 재할당이 항목 배열을 재배치할 수 있음에 유의하십시오. 따라서, 시퀀스가 변경될 수 없는 문맥에서만 하부 배열 포인터를 사용하십시오.

`PyObject* PySequence_ITEM (PyObject *o, Py_ssize_t i)`

Return value: New reference. *o*의 *i* 번째 요소를 반환하거나, 실패하면 NULL을 반환합니다. `PySequence_GetItem()`의 빠른 형식이지만, *o*에 대해 `PySequence_Check()`가 참인지 검사하지 않고, 음수 인덱스를 조정하지 않습니다.

7.4 매핑 프로토콜

PyObject_GetItem(), *PyObject_SetItem()* 및 *PyObject_DelItem()*도 참조하십시오.

int PyMapping_Check (PyObject *o)

객체가 매핑 프로토콜을 제공하거나 슬라이싱을 지원하면 1을 반환하고, 그렇지 않으면 0을 반환합니다. `__getitem__()` 메서드가 있는 파이썬 클래스의 경우 1을 반환한다는 점에 유의하십시오; 일반적으로 지원하는 키 형을 판단할 수 없기 때문입니다. 이 함수는 항상 성공합니다.

Py_ssize_t PyMapping_Size (PyObject *o)

Py_ssize_t PyMapping_Length (PyObject *o)

성공 시 객체 *o*의 키 수를 반환하고, 실패하면 -1을 반환합니다. 이는 파이썬 표현식 `len(o)`와 동등합니다.

PyObject* PyMapping_GetItemString (PyObject *o, const char *key)

Return value: New reference. 문자열 *key*에 해당하는 *o*의 요소나 실패 시 NULL을 반환합니다. 이는 파이썬 표현식 `o[key]`와 동등합니다. *PyObject_GetItem()*도 참조하십시오.

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

객체 *o*에서 문자열 *key*를 값 *v*에 매핑합니다. 실패하면 -1을 반환합니다. 이것은 파이썬 문장 `o[key] = v`와 동등합니다. *PyObject_SetItem()*도 참조하십시오. 이 함수는 *v*에 대한 참조를 훔치지 않습니다.

int PyMapping_DelItem (PyObject *o, PyObject *key)

객체 *o*에서 객체 *key*에 대한 매핑을 제거합니다. 실패하면 -1을 반환합니다. 이것은 파이썬 문장 `del o[key]`와 동등합니다. 이것은 *PyObject_DelItem()*의 별칭입니다.

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

객체 *o*에서 문자열 *key*에 대한 매핑을 제거합니다. 실패하면 -1을 반환합니다. 이것은 파이썬 문장 `del o[key]`와 동등합니다.

int PyMapping_HasKey (PyObject *o, PyObject *key)

매핑 객체에 *key* 키가 있으면 1을 반환하고, 그렇지 않으면 0을 반환합니다. 이는 파이썬 표현식 `key in o`와 동등합니다. 이 함수는 항상 성공합니다.

`__getitem__()` 메서드를 호출하는 동안 발생하는 예외는 억제됨에 유의하십시오. 에러 보고를 받으려면 대신 *PyObject_GetItem()*을 사용하십시오.

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

매핑 객체에 *key* 키가 있으면 1을 반환하고, 그렇지 않으면 0을 반환합니다. 이는 파이썬 표현식 `key in o`와 동등합니다. 이 함수는 항상 성공합니다.

`__getitem__()` 메서드를 호출하고 임시 문자열 객체를 만드는 동안 발생하는 예외는 억제됨에 유의하십시오. 에러 보고를 받으려면 대신 *PyMapping_GetItemString()*을 사용하십시오.

PyObject* PyMapping_Keys (PyObject *o)

Return value: New reference. 성공하면, 객체 *o*의 키 리스트를 반환합니다. 실패하면, NULL을 반환합니다.

버전 3.7에서 변경: 이전에는 함수가 리스트나 튜플을 반환했습니다.

PyObject* PyMapping_Values (PyObject *o)

Return value: New reference. 성공하면, 객체 *o*의 값 리스트를 반환합니다. 실패하면, NULL을 반환합니다.

버전 3.7에서 변경: 이전에는 함수가 리스트나 튜플을 반환했습니다.

PyObject* PyMapping_Items (PyObject *o)

Return value: New reference. 성공하면, 객체 *o*에 있는 항목 리스트를 반환합니다. 여기서 각 항목은 키-값 쌍을 포함하는 튜플입니다. 실패하면, NULL을 반환합니다.

버전 3.7에서 변경: 이전에는 함수가 리스트나 튜플을 반환했습니다.

7.5 이터레이터 프로토콜

특히 이터레이터를 사용하기 위한 두 함수가 있습니다.

`int PyIter_Check(PyObject *o)`

객체 *o*가 이터레이터 프로토콜을 지원하면 참을 돌려줍니다.

`PyObject* PyIter_Next(PyObject *o)`

Return value: New reference. 이터레이션 *o*에서 다음 값을 반환합니다. 객체는 이터레이터 여야 합니다 (이것을 확인하는 것은 호출자 책임입니다). 남은 값이 없으면, 예외가 설정되지 않은 상태로 NULL을 반환합니다. 항목을 꺼내는 동안 예외가 발생하면, NULL을 반환하고 예외를 전달합니다.

이터레이터를 이터레이트하는 루프를 작성하려면, C 코드는 이런 식으로 되어야 합니다:

```
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 */
}
```

7.6 버퍼 프로토콜

파이썬에서 사용할 수 있는 어떤 객체는 하부 메모리 배열 또는 버퍼에 대한 액세스를 감쌉니다. 이러한 객체에는 내장 `bytes`와 `bytearray`, 그리고 `array.array`와 같은 일부 확장형이 포함됩니다. 제삼자 라이브러리도 이미지 처리나 수치 해석과 같은 특수한 용도로 자체 형을 정의할 수 있습니다.

이러한 형은 각각 고유의 의미가 있지만, (아마도) 큰 메모리 버퍼에 의해 뒷받침되는 공통된 특징을 공유합니다. 어떤 상황에서는 중간 복사 없이 직접 버퍼에 액세스하는 것이 바람직합니다.

파이썬은 C 수준에서 **버퍼 프로토콜** 형식으로 이러한 기능을 제공합니다. 이 프로토콜에는 두 가지 측면이 있습니다:

- 생산자 측에서는, 형이 “버퍼 인터페이스”를 내보낼 수 있는데, 그 형의 객체가 하부 버퍼의 정보를 노출할 수 있게 합니다. 이 인터페이스는 **버퍼 객체 구조체** 절에서 설명됩니다.
- 소비자 측에서는, 객체의 원시 하부 데이터에 대한 포인터를 얻기 위해 여러 가지 방법을 사용할 수 있습니다(예를 들어 메서드 매개 변수).

`bytes`와 `bytearray`와 같은 간단한 객체는 하부 버퍼를 바이트 지향 형식으로 노출합니다. 다른 형태도 가능합니다; 예를 들어, `array.array`에 의해 노출되는 요소는 멀티 바이트 값이 될 수 있습니다.

버퍼 인터페이스의 소비자 예는 파일 객체의 `write()` 메서드입니다: 버퍼 인터페이스를 통해 일련의 바이트를 내보낼 수 있는 모든 객체는 파일에 기록될 수 있습니다. `write()`가 전달된 객체의 내부 내용에 대한 읽기 전용 액세스만 필요하지만, `readinto()`와 같은 다른 메서드는 인자의 내용에 쓰기 액세스가 필요합니다. 버퍼 인터페이스는 객체가 읽기-쓰기와 읽기 전용 버퍼를 선택적으로 허용하거나 거부할 수 있도록 합니다.

버퍼 인터페이스의 소비자가 대상 객체에 대해 버퍼를 얻는 방법에는 두 가지가 있습니다:

- 올바른 매개 변수로 `PyObject_GetBuffer()`를 호출합니다;
- `y*`, `w*` 또는 `s*` 형식 코드 중 하나를 사용하여 `PyArg_ParseTuple()`(또는 그 형제 중 하나)을 호출합니다.

두 경우 모두, 버퍼가 더는 필요하지 않으면 `PyBuffer_Release()`를 호출해야 합니다. 그렇게 하지 않으면 자원 누수와 같은 다양한 문제가 발생할 수 있습니다.

7.6.1 버퍼 구조체

버퍼 구조체(또는 단순히 “버퍼”)는 다른 객체의 바이너리 데이터를 파이썬 프로그래머에게 노출하는 방법으로 유용합니다. 또한, 복사 없는(zero-copy) 슬라이싱 메커니즘으로 사용할 수 있습니다. 메모리 블록을 참조하는 능력을 사용해서, 임의의 데이터를 파이썬 프로그래머에게 아주 쉽게 노출할 수 있습니다. 메모리는 C 확장의 큰 상수 배열일 수 있으며, 운영 체제 라이브러리로 전달되기 전에 조작하기 위한 원시 메모리 블록일 수도 있고, 네이티브 인 메모리(in-memory) 형식으로 구조화된 데이터를 전달하는 데 사용될 수도 있습니다.

파이썬 인터프리터가 노출하는 대부분의 데이터형과 달리, 버퍼는 `PyObject` 포인터가 아니라 단순한 C 구조체입니다. 이를 통해 매우 간단하게 만들고 복사할 수 있습니다. 버퍼를 감싸는 일반 래퍼가 필요할 때는, 메모리 뷰 객체를 만들 수 있습니다.

제공하는(exporting) 객체를 작성하는 간단한 지침은 버퍼 객체 구조체를 참조하십시오. 버퍼를 얻으려면, `PyObject_GetBuffer()`를 참조하십시오.

Py_buffer

void *buf

버퍼 필드에 의해 기술된 논리적 구조의 시작을 가리키는 포인터. 이것은 제공자(exporter)의 하부 물리적 메모리 블록 내의 모든 위치일 수 있습니다. 예를 들어, 음의 `strides`를 사용하면 값이 메모리 블록의 끝을 가리킬 수 있습니다.

연속 배열의 경우, 값은 메모리 블록의 시작을 가리킵니다.

void *obj

제공하는(exporting) 객체에 대한 새 참조. 참조는 소비자가 소유하고, `PyBuffer_Release()`에 의해 자동으로 감소하고 NULL로 설정됩니다. 이 필드는 표준 C-API 함수의 반환 값과 동등합니다.

특수한 경우로, `PyMemoryView_FromBuffer()` 나 `PyBuffer_FillInfo()`로 감싸진 임시(temporary) 버퍼의 경우, 이 필드는 NULL입니다. 일반적으로, 제공하는(exporting) 객체는 이 체계를 사용하지 않아야 합니다.

Py_ssize_t len

`product(shape) * itemsize`. 연속 배열의 경우, 하부 메모리 블록의 길이입니다. 불연속 배열의 경우, 연속 표현으로 복사된다면 논리적 구조체가 갖게 될 길이입니다.

`((char *)buf)[0]`에서 `((char *)buf)[len-1]` 범위의 액세스는 연속성을 보장하는 요청으로 버퍼가 확보된 경우에만 유효합니다. 대부분 이러한 요청은 `PyBUF_SIMPLE` 또는 `PyBUF_WRITABLE`입니다.

int readonly

버퍼가 읽기 전용인지를 나타내는 표시기입니다. 이 필드는 `PyBUF_WRITABLE` 플래그로 제어됩니다.

Py_ssize_t *itemsize*

단일 요소의 항목 크기(바이트)입니다. NULL이 아닌 *format* 값에 호출된 `struct.calcsize()` 값과 같습니다.

중요한 예외: 소비자가 *PyBUF_FORMAT* 플래그 없이 버퍼를 요청하면, *format*은 NULL로 설정되지만, *itemsize*는 여전히 원래 형식의 값을 갖습니다.

*shape*이 있으면, `product(shape) * itemsize == len` 동치가 계속 성립하고 소비자는 *itemsize*를 사용하여 버퍼를 탐색할 수 있습니다.

PyBUF_SIMPLE 이나 *PyBUF_WRITABLE* 요청의 결과로 *shape*이 NULL이면, 소비자는 *itemsize*를 무시하고 `itemsize == 1`로 가정해야 합니다.

const char **format*

단일 항목의 내용을 설명하는 struct 모듈 스타일 문법의 NUL 종료 문자열. 이것이 NULL이면, "B"(부호 없는 바이트)를 가정합니다.

이 필드는 *PyBUF_FORMAT* 플래그로 제어됩니다.

int *ndim*

메모리가 *n* 차원 배열로 나타내는 차원 수. 0이면, *buf*는 스칼라를 나타내는 단일 항목을 가리킵니다. 이 경우, *shape*, *strides* 및 *suboffsets*는 반드시 NULL 이어야 합니다.

매크로 *PyBUF_MAX_NDIM*는 최대 차원 수를 64로 제한합니다. 제공자는 이 제한을 존중해야 하며, 다차원 버퍼의 소비자는 *PyBUF_MAX_NDIM* 차원까지 처리할 수 있어야 합니다.

Py_ssize_t **shape*

n-차원 배열로 메모리의 모양을 나타내는 길이 *ndim*의 *Py_ssize_t* 배열. `shape[0] * ... * shape[ndim-1] * itemsize`는 *len*과 같아야 합니다.

모양 값은 `shape[n] >= 0`로 제한됩니다. `shape[n] == 0`인 경우는 특별한 주의가 필요합니다. 자세한 정보는 복잡한 배열을 참조하십시오.

shape 배열은 소비자에게 읽기 전용입니다.

Py_ssize_t **strides*

각 차원에서 새 요소를 가져오기 위해 건너뛸 바이트 수를 제공하는 길이 *ndim*의 *Py_ssize_t* 배열.

스트라이드 값은 임의의 정수일 수 있습니다. 일반 배열의 경우, 스트라이드는 보통 양수이지만, 소비자는 `strides[n] <= 0`인 경우를 처리할 수 있어야 합니다. 자세한 내용은 복잡한 배열을 참조하십시오.

strides 배열은 소비자에게 읽기 전용입니다.

Py_ssize_t **suboffsets*

길이 *ndim*의 *Py_ssize_t* 배열. `suboffsets[n] >= 0` 면, *n* 번째 차원을 따라 저장된 값은 포인터이고 서브 오프셋 값은 역참조(de-referencing) 후 각 포인터에 더할 바이트 수를 나타냅니다. 음의 서브 오프셋 값은 역참조(de-referencing)가 발생하지 않아야 함을 나타냅니다(연속 메모리 블록에서의 스트라이드).

모든 서브 오프셋이 음수면(즉, 역참조가 필요하지 않으면), 이 필드는 NULL(기본값) 이어야 합니다.

이 유형의 배열 표현은 파이썬 이미징 라이브러리(PIL)에서 사용됩니다. 이러한 배열 요소에 액세스하는 방법에 대한 자세한 내용은 복잡한 배열을 참조하십시오.

suboffsets 배열은 소비자에게 읽기 전용입니다.

void **internal*

이것은 제공하는(exporting) 객체에 의해 내부적으로 사용됩니다. 예를 들어, 이것은 제공자(exporter)가 정수로 다시 캐스팅할 수 있으며, 버퍼가 해제될 때 *shape*, *strides* 및 *suboffsets* 배열을 해제해야 하는지에 대한 플래그를 저장하는 데 사용됩니다. 소비자가 이 값을 변경해서는 안 됩니다.

7.6.2 버퍼 요청 유형

버퍼는 대개 `PyObject_GetBuffer()`를 통해 제공하는(exporting) 객체로 버퍼 요청을 보내서 얻습니다. 메모리의 논리적 구조의 복잡성이 크게 다를 수 있으므로, 소비자는 처리할 수 있는 정확한 버퍼 유형을 지정하기 위해 *flags* 인자를 사용합니다.

모든 *Py_buffer* 필드는 요청 유형에 의해 모호하지 않게 정의됩니다.

요청 독립적 필드

다음 필드는 *flags*의 영향을 받지 않고 항상 올바른 값으로 채워져야 합니다: *obj*, *buf*, *len*, *itemsize*, *ndim*.

readonly, format

PyBUF_WRITABLE

readonly 필드를 제어합니다. 설정되면, 제공자는 반드시 쓰기 가능한 버퍼를 제공하거나 실패를 보고해야 합니다. 그렇지 않으면, 제공자는 읽기 전용 버퍼나 쓰기 가능 버퍼를 제공할 수 있지만, 모든 소비자에 대해 일관성을 유지해야 합니다.

PyBUF_FORMAT

format 필드를 제어합니다. 설정되면, 이 필드를 올바르게 채워야 합니다. 그렇지 않으면, 이 필드는 반드시 NULL 이어야 합니다.

*PyBUF_WRITABLE*은 다음 섹션의 모든 플래그와 I 될 수 있습니다. *PyBUF_SIMPLE*이 0으로 정의되므로, *PyBUF_WRITABLE*은 독립형 플래그로 사용되어 간단한 쓰기 가능한 버퍼를 요청할 수 있습니다.

*PyBUF_FORMAT*은 *PyBUF_SIMPLE*을 제외한 임의의 플래그와 I 될 수 있습니다. *PyBUF_SIMPLE*은 이미 형식 B(부호 없는 바이트)를 의미합니다.

shape, strides, suboffsets

메모리의 논리 구조를 제어하는 플래그는 복잡도가 감소하는 순서로 나열됩니다. 각 플래그는 그 아래에 있는 플래그의 모든 비트를 포함합니다.

요청	shape	strides	suboffsets
PyBUF_INDIRECT	yes	yes	필요하면
PyBUF_STRIDES	yes	yes	NULL
PyBUF_ND	yes	NULL	NULL
PyBUF_SIMPLE	NULL	NULL	NULL

연속성 요청

C 나 포트란 연속성을 명시적으로 요청할 수 있는데, 스트라이드 정보를 포함하기도 그렇지 않기도 합니다. 스트라이드 정보가 없으면, 버퍼는 C-연속이어야 합니다.

요청	shape	strides	suboffsets	연속성
PyBUF_C_CONTIGUOUS	yes	yes	NULL	C
PyBUF_F_CONTIGUOUS	yes	yes	NULL	F
PyBUF_ANY_CONTIGUOUS	yes	yes	NULL	C 또는 F
<i>PyBUF_ND</i>	yes	NULL	NULL	C

복합 요청

모든 가능한 요청은 앞 절의 플래그 조합에 의해 완전히 정의됩니다. 편의상, 버퍼 프로토콜은 자주 사용되는 조합을 단일 플래그로 제공합니다.

다음 표에서 *U*는 정의되지 않은 연속성을 나타냅니다. 소비자는 연속성을 판단하기 위해 *PyBuffer_IsContiguous()*를 호출해야 합니다.

요청	shape	strides	suboffsets	연속성	readonly	format
PyBUF_FULL	yes	yes	필요하면	U	0	yes
PyBUF_FULL_RO	yes	yes	필요하면	U	1 또는 0	yes
PyBUF_RECORDS	yes	yes	NULL	U	0	yes
PyBUF_RECORDS_RO	yes	yes	NULL	U	1 또는 0	yes
PyBUF_STRIDED	yes	yes	NULL	U	0	NULL
PyBUF_STRIDED_RO	yes	yes	NULL	U	1 또는 0	NULL
PyBUF_CONTIG	yes	NULL	NULL	C	0	NULL
PyBUF_CONTIG_RO	yes	NULL	NULL	C	1 또는 0	NULL

7.6.3 복잡한 배열

NumPy-스타일: `shape`과 `strides`

NumPy 스타일 배열의 논리적 구조는 `itemsize`, `ndim`, `shape` 및 `strides`로 정의됩니다.

`ndim == 0`이면, `buf`가 가리키는 메모리 위치가 `itemsize` 크기의 스칼라로 해석됩니다. 이 경우, `shape`와 `strides`는 모두 NULL입니다.

`strides`가 NULL이면, 배열은 표준 n-차원 C 배열로 해석됩니다. 그렇지 않으면, 소비자는 다음과 같이 n-차원 배열에 액세스해야 합니다:

```
ptr = (char *)buf + indices[0] * strides[0] + ... + indices[n-1] * strides[n-1];
item = *((typeof(item) *)ptr);
```

위에서 언급했듯이, `buf`는 실제 메모리 블록 내의 모든 위치를 가리킬 수 있습니다. 제공자(exporter)는 이 함수로 버퍼의 유효성을 검사할 수 있습니다:

```
def verify_structure(memlen, itemsize, ndim, shape, strides, offset):
    """Verify that the parameters represent a valid array within
    the bounds of the allocated memory:
        char *mem: start of the physical memory block
        memlen: length of the physical memory block
        offset: (char *)buf - mem
    """
    if offset % itemsize:
        return False
    if offset < 0 or offset+itemsize > memlen:
        return False
    if any(v % itemsize for v in strides):
        return False

    if ndim <= 0:
        return ndim == 0 and not shape and not strides
    if 0 in shape:
        return True

    imin = sum(strides[j]*(shape[j]-1) for j in range(ndim)
               if strides[j] <= 0)
    imax = sum(strides[j]*(shape[j]-1) for j in range(ndim)
               if strides[j] > 0)

    return 0 <= offset+imin and offset+imax+itemsize <= memlen
```

PIL-스타일: `shape`, `strides` 및 `suboffsets`

일반 항목 외에도, PIL 스타일 배열에는 차원의 다음 요소를 가져오기 위해 따라야 하는 포인터가 포함될 수 있습니다. 예를 들어, 일반 3-차원 C 배열 `char v[2][2][3]`는 2개의 2-차원 배열을 가리키는 2개의 포인터 배열로 볼 수도 있습니다: `char (*v[2])[2][3]`. `suboffsets` 표현에서, 이 두 포인터는 `buf`의 시작 부분에 임베드될 수 있는데, 메모리의 어느 위치 에나 배치될 수 있는 두 개의 `char x[2][3]` 배열을 가리킵니다.

다음은 NULL이 아닌 `strides`와 `suboffsets`가 있을 때, N-차원 인덱스가 가리키는 N-차원 배열의 요소에 대한 포인터를 반환하는 함수입니다:

```
void *get_item_pointer(int ndim, void *buf, Py_ssize_t *strides,
                      Py_ssize_t *suboffsets, Py_ssize_t *indices) {
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```

char *pointer = (char*)buf;
int i;
for (i = 0; i < ndim; i++) {
    pointer += strides[i] * indices[i];
    if (suboffsets[i] >= 0) {
        pointer = *((char**)pointer) + suboffsets[i];
    }
}
return (void*)pointer;
}

```

7.6.4 버퍼 관련 함수

int PyObject_CheckBuffer (*PyObject *obj*)

*obj*가 버퍼 인터페이스를 지원하면 1을 반환하고, 그렇지 않으면 0을 반환합니다. 1이 반환될 때, *PyObject_GetBuffer()*가 성공할 것이라고 보장하지는 않습니다. 이 함수는 항상 성공합니다.

int PyObject_GetBuffer (*PyObject *exporter, Py_buffer *view, int flags*)

*flags*에 지정된 대로 *view*를 채우도록 *exporter*에게 요청을 보냅니다. 제공자(*exporter*)가 정확한 유형의 버퍼를 제공할 수 없다면, *PyExc_BufferError*를 일으키고, *view->obj*를 NULL로 설정하고, -1을 반환해야 합니다.

성공하면, *view*를 채우고, *view->obj*를 *exporter*에 대한 새 참조로 설정하고, 0을 반환합니다. 요청을 단일 객체로 리디렉션하는 연결된(chained) 버퍼 공급자의 경우, *view->obj*는 *exporter* 대신 이 객체를 참조할 수 있습니다(버퍼 객체 구조체를 보세요).

*PyObject_GetBuffer()*에 대한 성공적인 호출은 *PyBuffer_Release()*에 대한 호출과 쌍을 이루어야 합니다, *malloc()*과 *free()*와 유사합니다. 따라서, 소비자가 버퍼로 작업한 후에는, *PyBuffer_Release()*를 정확히 한 번 호출해야 합니다.

void PyBuffer_Release (*Py_buffer *view*)

버퍼 *view*를 해제하고 *view->obj*에 대한 참조 횟수를 감소시킵니다. 버퍼가 더는 사용되지 않을 때, 이 함수를 반드시 호출해야 합니다. 그렇지 않으면 참조 누수가 발생할 수 있습니다.

*PyObject_GetBuffer()*를 통해 얻지 않은 버퍼에 이 함수를 호출하는 것은 에러입니다.

Py_ssize_t PyBuffer_SizeFromFormat (const char *)

*format*이 암시하는 *itemsizes*를 반환합니다. 이 함수는 아직 구현되지 않았습니다.

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

*view*로 정의된 메모리가 C 스타일(*order*가 'C')이나 포트란 스타일(*order*가 'F') 연속이거나 둘 중 하나(*order*가 'A')면 1을 반환합니다. 그렇지 않으면 0을 반환합니다. 이 함수는 항상 성공합니다.

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

주어진 *view* 내부의 *indices*가 가리키는 메모리 영역을 가져옵니다. *indices*는 *view->ndim* 인덱스의 배열을 가리켜야 합니다.

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

*buf*에 있는 연속된 *len* 바이트를 *view*로 복사합니다. *fort*는 'C' 또는 'F'(C 스타일 또는 포트란 스타일 순서)일 수 있습니다. 성공하면 0이 반환되고, 에러가 있으면 -1이 반환됩니다.

int PyBuffer_ToContiguous (void *buf, *Py_buffer *src, Py_ssize_t len, char order*)

*src*에 있는 *len* 바이트를 *buf*에 연속 표현으로 복사합니다. *order*는 'C' 또는 'F' 또는 'A'(C 스타일 또는 포트란 스타일 순서 또는 둘 중 하나)일 수 있습니다. 성공하면 0이 반환되고, 에러가 있으면 -1이 반환됩니다.

이 함수는 *len != src->len*이면 실패합니다.

`void PyBuffer_FillContiguousStrides` (`int ndims`, `Py_ssize_t *shape`, `Py_ssize_t *strides`, `int itemsize`, `char order`)
`strides` 배열을 주어진 요소당 바이트 수와 주어진 `shape` 으로 연속 (`order`가 'C'면 C 스타일, `order`가 'F'면 포트란 스타일) 배열의 바이트 스트라이드로 채웁니다.

`int PyBuffer_FillInfo` (`Py_buffer *view`, `PyObject *exporter`, `void *buf`, `Py_ssize_t len`, `int readonly`, `int flags`)
`readonly`에 따라 쓰기 가능성이 설정된 `len` 크기의 `buf`를 노출하려는 제공자(`exporter`)에 대한 버퍼 요청을 처리합니다. `buf`는 부호 없는 바이트의 시퀀스로 해석됩니다.

`flags` 인자는 요청 유형을 나타냅니다. 이 함수는 `buf`가 읽기 전용으로 지정되고 `PyBUF_WRITABLE`이 `flags`에 설정되어 있지 않으면, 항상 플래그가 지정하는 대로 `view`를 채웁니다.

성공하면, `view->obj`를 `exporter`에 대한 새 참조로 설정하고, 0을 반환합니다. 그렇지 않으면, `PyExc_BufferError`를 일으키고, `view->obj`를 NULL로 설정한 다음 -1을 반환합니다.

이 함수가 `getbufferproc`의 일부로 사용되면, `exporter`가 제공하는(`exporting`) 객체로 설정되어야 하고, `flags`는 수정되지 않은 채로 전달되어야 합니다. 그렇지 않으면 `exporter`가 NULL이어야 합니다.

7.7 낚은 버퍼 프로토콜

버전 3.0부터 폐지.

이 함수는 파이썬 2에서 “낚은 버퍼 프로토콜” API 일부분이었습니다. 파이썬 3에서는 이 프로토콜이 더는 존재하지 않지만 2.x 코드 이식을 쉽게 하도록 함수들은 여전히 노출됩니다. 이들은 새 버퍼 프로토콜을 둘러싼 호환성 래퍼 역할을 하지만, 버퍼를 제공할 때 얻은 자원의 수명을 제어할 수는 없습니다.

따라서, `PyObject_GetBuffer()`(또는 `y*` 나 `w*` 포맷 코드를 사용하는 `PyArg_ParseTuple()` 계열의 함수)를 호출하여 개체에 대한 버퍼 뷰를 가져오고, 버퍼 뷰를 해제할 수 있을 때 `PyBuffer_Release()`를 호출하는 것이 좋습니다.

`int PyObject_AsCharBuffer` (`PyObject *obj`, `const char **buffer`, `Py_ssize_t *buffer_len`)
문자 기반 입력으로 사용할 수 있는 읽기 전용 메모리 위치에 대한 포인터를 반환합니다. `obj` 인자는 단일 세그먼트 문자 버퍼 인터페이스를 지원해야 합니다. 성공하면, 0을 반환하고, `buffer`를 메모리 위치로 설정하고, `buffer_len`을 버퍼 길이로 설정합니다. 에러 시에, -1을 반환하고, `TypeError`를 설정합니다.

`int PyObject_AsReadBuffer` (`PyObject *obj`, `const void **buffer`, `Py_ssize_t *buffer_len`)
임의의 데이터를 포함하는 읽기 전용 메모리 위치에 대한 포인터를 반환합니다. `obj` 인자는 단일 세그먼트 읽기 가능 버퍼 인터페이스를 지원해야 합니다. 성공하면, 0을 반환하고, `buffer`를 메모리 위치로 설정하고, `buffer_len`을 버퍼 길이로 설정합니다. 에러 시에, -1을 반환하고, `TypeError`를 설정합니다.

`int PyObject_CheckReadBuffer` (`PyObject *o`)
`o`가 단일 세그먼트 읽기 가능 버퍼 인터페이스를 지원하면 1을 반환합니다. 그렇지 않으면, 0을 반환합니다. 이 함수는 항상 성공합니다.

이 함수는 버퍼를 가져오고 해제하려고 하며, 해당 함수를 호출하는 동안 발생하는 예외는 억제됨에 유의하십시오. 에러 보고를 받으려면 대신 `PyObject_GetBuffer()`를 사용하십시오.

`int PyObject_AsWriteBuffer` (`PyObject *obj`, `void **buffer`, `Py_ssize_t *buffer_len`)
쓰기 가능한 메모리 위치에 대한 포인터를 반환합니다. `obj` 인자는 단일 세그먼트, 문자 버퍼 인터페이스를 지원해야 합니다. 성공하면, 0을 반환하고, `buffer`를 메모리 위치로 설정하고, `buffer_len`을 버퍼 길이로 설정합니다. 에러 시에, -1을 반환하고, `TypeError`를 설정합니다.

이 장의 함수는 특정 파이썬 객체 형에게만 적용됩니다. 그들에게 잘못된 형의 객체를 전달하는 것은 좋은 생각이 아닙니다; 파이썬 프로그램에서 객체를 받았는데 올바른 형을 가졌는지 확실하지 않다면, 먼저 형 검사를 수행해야 합니다; 예를 들어, 객체가 딕셔너리인지 확인하려면, `PyDict_Check()`를 사용하십시오. 이 장은 파이썬 객체 형의 “족보”처럼 구성되어 있습니다.

경고: 이 장에서 설명하는 함수는 전달되는 객체의 형을 주의 깊게 검사하지만, 많은 함수는 유효한 객체 대신 전달되는 NULL을 확인하지 않습니다. NULL을 전달하면 메모리 액세스 위반이 발생하고 인터프리터가 즉시 종료될 수 있습니다.

8.1 기본 객체

이 절에서는 파이썬 형 객체와 싱글톤 객체 None에 대해 설명합니다.

8.1.1 형 객체

PyObject

내장형을 기술하는 데 사용되는 객체의 C 구조체.

*PyObject** **PyType_Type**

이것은 형 객체의 형 객체입니다; 파이썬 계층의 `type`과 같은 객체입니다.

int **PyType_Check** (*PyObject* *o)

객체 *o*가 표준형 객체에서 파생된 형의 인스턴스를 포함하여 형 객체면 참을 반환합니다. 다른 모든 경우 거짓을 반환합니다.

int **PyType_CheckExact** (*PyObject* *o)

객체 *o*가 형 객체이지만, 표준형 객체의 서브 형이 아니면 참을 반환합니다. 다른 모든 경우 거짓을 반환합니다.

unsigned int **PyType_ClearCache** ()

내부 조회 캐시를 지웁니다. 현재의 버전 태그를 반환합니다.

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

type의 *tp_flags* 멤버를 반환합니다. 이 함수는 주로 *Py_LIMITED_API*와 함께 사용하기 위한 것입니다; 개별 플래그 비트는 파이썬 배포 간에 안정적인 것으로 보장되지만, *tp_flags* 자체에 대한 액세스는 제한된 API 일부가 아닙니다.

버전 3.2에 추가.

버전 3.4에서 변경: 반환형은 이제 long이 아니라 unsigned long입니다.

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

형과 그것의 모든 서브 형에 대한 내부 검색 캐시를 무효로 합니다. 형의 어트리뷰트나 베이스 클래스를 수동으로 수정한 후에는 이 함수를 호출해야 합니다.

int **PyType_HasFeature** (*PyTypeObject** o, int feature)

형 객체 o가 기능 feature를 설정하면 참을 반환합니다. 형 기능은 단일 비트 플래그로 표시됩니다.

int **PyType_IS_GC** (*PyTypeObject** o)

형 객체가 순환 검출기에 대한 지원을 포함하고 있으면 참을 반환합니다. 이것은 형 플래그 *Py_TPFLAGS_HAVE_GC*를 검사합니다.

int **PyType_IsSubtype** (*PyTypeObject** a, *PyTypeObject** b)

a가 b의 서브 형이면 참을 반환합니다.

이 함수는 실제 서브 형만 검사합니다. 즉, *__subclasscheck__*()가 b에 대해 호출되지 않습니다. *issubclass*()가 수행하는 것과 같은 검사를 하려면 *PyObject_IsSubclass*()를 호출하십시오.

*PyObject** **PyType_GenericAlloc** (*PyTypeObject** type, Py_ssize_t nitems)

Return value: New reference. 형 객체의 *tp_alloc* 슬롯을 위한 일반 처리기. 파이썬의 기본 메모리 할당 메커니즘을 사용하여 새 인스턴스를 할당하고 모든 내용을 NULL로 초기화합니다.

*PyObject** **PyType_GenericNew** (*PyTypeObject** type, *PyObject** args, *PyObject** kwds)

Return value: New reference. 형 객체의 *tp_new* 슬롯을 위한 일반 처리기. 형의 *tp_alloc* 슬롯을 사용하여 새 인스턴스를 만듭니다.

int **PyType_Ready** (*PyTypeObject** type)

형 객체를 마무리합니다. 초기화를 완료하려면 모든 형 객체에 대해 이 메서드를 호출해야 합니다. 이 함수는 형의 베이스 클래스에서 상속된 슬롯을 추가합니다. 성공 시 0을 반환하고, 오류 시 -1을 반환하고 예외를 설정합니다.

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

지정된 슬롯에 저장된 함수 포인터를 반환합니다. 결과가 NULL이면, 슬롯이 NULL이거나 함수가 유효하지 않은 매개 변수로 호출되었음을 나타냅니다. 호출자는 일반적으로 결과 포인터를 적절한 함수 형으로 캐스팅합니다.

slot 인자의 가능한 값은 *PyType_Slot.slot*을 참조하십시오.

type이 힙 형이 아니면 예외가 발생합니다.

버전 3.4에 추가.

힙에 할당된 형 만들기

다음 함수와 구조체는 힙 형을 만드는 데 사용됩니다.

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

Return value: New reference. spec으로 힙 형 객체를 만들고 반환합니다 (*Py_TPFLAGS_HEAPTYPE*).

bases가 튜플이면, 생성된 힙 형에는 그것에 포함된 모든 형이 베이스형으로 포함됩니다.

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.

이 함수는 새로운 형에 *PyType_Ready()* 를 호출합니다.

버전 3.3에 추가.

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

Return value: New reference. *PyType_FromSpecWithBases*(spec, NULL) 와 동등합니다.

PyType_Spec

형의 행동을 정의하는 구조체.

const char* **PyType_Spec.name**

형의 이름, *PyObject.tp_name*을 설정하는 데 사용됩니다.

int **PyType_Spec.basicsize**

int **PyType_Spec.itemsize**

인스턴스의 크기 (바이트), *PyObject.tp_basicsize*와 *PyObject.tp_itemsize*를 설정하는 데 사용됩니다.

int **PyType_Spec.flags**

형 플래그, *PyObject.tp_flags*를 설정하는 데 사용됩니다.

Py_TPFLAGS_HEAPTYPE 플래그가 설정되어 있지 않으면, *PyType_FromSpecWithBases()* 가 자동으로 플래그를 설정합니다.

PyType_Slot ***PyType_Spec.slots**

PyType_Slot 구조체의 배열. 특수 슬롯값 {0, NULL}에 의해 종료됩니다.

PyType_Slot

형의 선택적 기능을 정의하는 구조체, 슬롯 ID와 값 포인터를 포함합니다.

int **PyType_Slot.slot**

슬롯 ID.

슬롯 ID는 구조체 *PyObject*, *PyNumberMethods*, *PySequenceMethods*, *PyMappingMethods* 및 *PyAsyncMethods*의 필드 이름에 *Py_* 접두사를 붙인 이름을 사용합니다. 예를 들어, :

- *PyObject.tp_dealloc*을 설정하는 *Py_tp_dealloc*
- *PyNumberMethods.nb_add*를 설정하는 *Py_nb_add*
- *PySequenceMethods.sq_length*를 설정하는 *Py_sq_length*

다음 필드는 *PyType_Spec*과 *PyType_Slot*을 사용하여 설정할 수 없습니다:

- *tp_dict*
- *tp_mro*
- *tp_cache*
- *tp_subclasses*

- `tp_weaklist`
- `tp_print`
- `tp_weaklistoffset`
- `tp_dictoffset`
- `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.

void ***PyType_Slot.pfunc**

슬롯의 원하는 값입니다. 대부분 이것은 함수에 대한 포인터입니다.

NULL이 아닐 수 있습니다.

8.1.2 None 객체

None에 대한 `PyObject`는 파이썬/C API에서 직접 노출되지 않습니다. None은 싱글톤이기 때문에 (C에서 `==`를 사용해서) 객체 아이덴티티를 검사하는 것으로 충분합니다. 같은 이유로 `PyNone_Check()` 함수가 없습니다.

*PyObject** **Py_None**

값의 부재를 나타내는 파이썬 None 객체입니다. 이 객체에는 메서드가 없습니다. 참조 횟수와 관련하여 다른 객체와 마찬가지로 처리해야 합니다.

Py_RETURN_NONE

C 함수 내에서 `Py_None`를 반환하는 것을 올바르게 처리합니다 (즉, None의 참조 횟수를 증가시키고 반환합니다).

8.2 숫자 객체

8.2.1 정수 객체

모든 정수는 임의의 크기의 “long” 정수 객체로 구현됩니다.

예러 시, 대부분의 `PyLong_As*` API는 숫자와 구별할 수 없는 (return type) -1을 반환합니다. 모호성을 제거하려면 `PyErr_Occurred()`를 사용하십시오.

PyLongObject

이 `PyObject`의 서브 형은 파이썬 정수 객체를 나타냅니다.

PyObject **PyLong_Type**

이 `PyObject` 인스턴스는 파이썬 정수 형을 나타냅니다. 이것은 파이썬 계층의 `int`와 같은 객체입니다.

int **PyLong_Check** (*PyObject* *p)

인자가 `PyLongObject`이나 `PyLongObject`의 서브 형이면 참을 반환합니다.

int **PyLong_CheckExact** (*PyObject* *p)

인자가 `PyLongObject`이지만 `PyLongObject`의 서브 형이 아니면 참을 반환합니다.

*PyObject** **PyLong_FromLong** (long v)

Return value: New reference. v로부터 새 `PyLongObject` 객체를 반환하거나, 실패하면 NULL을 반환합니다.

현재 구현은 -5와 256 사이의 모든 정수에 대해 정수 객체의 배열을 유지합니다. 이 범위에 있는 정수를 만들면 실제로는 기존 객체에 대한 참조만 반환됩니다. 따라서 1의 값을 변경하는 것이 가능합니다. 이때 파이썬의 동작은 정의되지 않은 것으로 판단됩니다. :-)

PyObject* **PyLong_FromUnsignedLong** (unsigned long *v*)

Return value: New reference. C unsigned long으로부터 새 *PyLongObject* 객체를 반환하거나, 실패하면 NULL을 반환합니다.

PyObject* **PyLong_FromSsize_t** (Py_ssize_t *v*)

Return value: New reference. C Py_ssize_t로부터 새 *PyLongObject* 객체를 반환하거나, 실패하면 NULL을 반환합니다.

PyObject* **PyLong_FromSize_t** (size_t *v*)

Return value: New reference. C size_t로부터 새 *PyLongObject* 객체를 반환하거나, 실패하면 NULL을 반환합니다.

PyObject* **PyLong_FromLongLong** (long long *v*)

Return value: New reference. C long long으로부터 새 *PyLongObject* 객체를 반환하거나, 실패하면 NULL을 반환합니다.

PyObject* **PyLong_FromUnsignedLongLong** (unsigned long long *v*)

Return value: New reference. C unsigned long long으로부터 새 *PyLongObject* 객체를 반환하거나, 실패하면 NULL을 반환합니다.

PyObject* **PyLong_FromDouble** (double *v*)

Return value: New reference. *v*의 정수 부분으로부터 새 *PyLongObject* 객체를 반환하거나, 실패하면 NULL을 반환합니다.

PyObject* **PyLong_FromString** (const char **str*, char ***pend*, int *base*)

Return value: New reference. *str*의 문자열 값을 기반으로 한 새 *PyLongObject*를 반환합니다. 문자열 값은 *base*의 진수(기수)에 따라 해석됩니다. *pend*가 NULL이 아니면, **pend*는 숫자 표현의 뒤에 오는 첫 번째 문자를 가리킵니다. *base*가 0이면, *str*은 integers 정의를 사용해서 해석됩니다; 이때, 0이 아닌 십진수의 선행 0은 ValueError를 발생시킵니다. *base*가 0이 아니면, 2와 36 사이에 있어야 하며, 경계를 포함합니다. 선행 공백과 진수 지정자 뒤나 숫자 사이의 단일 밑줄은 무시됩니다. 숫자가 없으면 ValueError가 발생합니다.

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

Return value: New reference. 유니코드 숫자의 시퀀스를 파이썬 정숫값으로 변환합니다.

Deprecated since version 3.3, will be removed in version 3.10: 이전 스타일의 *Py_UNICODE* API의 일부; *PyLong_FromUnicodeObject()*를 사용하는 것으로 변경하십시오.

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

Return value: New reference. 문자열 *u*에 있는 유니코드 숫자의 시퀀스를 파이썬 정숫값으로 변환합니다.

버전 3.3에 추가.

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

Return value: New reference. 포인터 *p*로부터 파이썬 정수를 만듭니다. 포인터 값은 *PyLong_AsVoidPtr()*를 사용하여 결괏값에서 조회할 수 있습니다.

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

*obj*의 C long 표현을 반환합니다. *obj*가 *PyLongObject*의 인스턴스가 아니면, (있다면) 먼저 *__index__()*나 *__int__()* 메서드를 호출하여 *PyLongObject*로 변환합니다.

*obj*의 값이 long의 범위를 벗어나면 OverflowError를 발생시킵니다.

예러 시 -1을 반환합니다. 모호성을 제거하려면 *PyErr_Occurred()*를 사용하십시오.

버전 3.8에서 변경: 사용할 수 있으면 *__index__()*를 사용합니다.

버전 3.8부터 폐지: *__int__()* 사용은 폐지되었습니다.

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

obj의 C long 표현을 반환합니다. obj가 *PyLongObject*의 인스턴스가 아니면, (있다면) 먼저 `__index__()` 나 `__int__()` 메서드를 호출하여 *PyLongObject*로 변환합니다.

obj의 값이 `LONG_MAX`보다 크거나 `LONG_MIN`보다 작으면, *overflow를 각각 1이나 -1로 설정하고 -1을 반환합니다; 그렇지 않으면, *overflow를 0으로 설정합니다. 다른 예외가 발생하면 *overflow를 0으로 설정하고 -1을 평소와 같이 반환합니다.

에러 시 -1을 반환합니다. 모호성을 제거하려면 *PyErr_Occurred()*를 사용하십시오.

버전 3.8에서 변경: 사용할 수 있으면 `__index__()`를 사용합니다.

버전 3.8부터 폐지: `__int__()` 사용은 폐지되었습니다.

long long PyLong_AsLongLong (*PyObject* *obj)

obj의 C long long 표현을 반환합니다. obj가 *PyLongObject*의 인스턴스가 아니면, (있다면) 먼저 `__index__()` 나 `__int__()` 메서드를 호출하여 *PyLongObject*로 변환합니다.

obj의 값이 long long의 범위를 벗어나면 *OverflowError*를 발생시킵니다.

에러 시 -1을 반환합니다. 모호성을 제거하려면 *PyErr_Occurred()*를 사용하십시오.

버전 3.8에서 변경: 사용할 수 있으면 `__index__()`를 사용합니다.

버전 3.8부터 폐지: `__int__()` 사용은 폐지되었습니다.

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

obj의 C long long 표현을 반환합니다. obj가 *PyLongObject*의 인스턴스가 아니면, (있다면) 먼저 `__index__()` 나 `__int__()` 메서드를 호출하여 *PyLongObject*로 변환합니다.

obj의 값이 `PY_LLONG_MAX`보다 크거나 `PY_LLONG_MIN`보다 작으면, *overflow를 각각 1이나 -1로 설정하고 -1을 반환합니다; 그렇지 않으면, *overflow를 0으로 설정합니다. 다른 예외가 발생하면 *overflow를 0으로 설정하고 -1을 평소와 같이 반환합니다.

에러 시 -1을 반환합니다. 모호성을 제거하려면 *PyErr_Occurred()*를 사용하십시오.

버전 3.2에 추가.

버전 3.8에서 변경: 사용할 수 있으면 `__index__()`를 사용합니다.

버전 3.8부터 폐지: `__int__()` 사용은 폐지되었습니다.

Py_ssize_t PyLong_AsSsize_t (*PyObject* *pylong)

pylong의 C *Py_ssize_t* 표현을 반환합니다. pylong은 *PyLongObject*의 인스턴스여야 합니다.

pylong의 값이 *Py_ssize_t*의 범위를 벗어나면 *OverflowError*를 발생시킵니다.

에러 시 -1을 반환합니다. 모호성을 제거하려면 *PyErr_Occurred()*를 사용하십시오.

unsigned long PyLong_AsUnsignedLong (*PyObject* *pylong)

pylong의 C unsigned long 표현을 반환합니다. pylong은 *PyLongObject*의 인스턴스여야 합니다.

pylong의 값이 unsigned long의 범위를 벗어나면 *OverflowError*를 발생시킵니다.

에러 시 (unsigned long) -1을 반환합니다. 모호성을 제거하려면 *PyErr_Occurred()*를 사용하십시오.

size_t PyLong_AsSize_t (*PyObject* *pylong)

pylong의 C *size_t* 표현을 반환합니다. pylong은 *PyLongObject*의 인스턴스여야 합니다.

pylong의 값이 *size_t*의 범위를 벗어나면 *OverflowError*를 발생시킵니다.

에러 시 (size_t) -1을 반환합니다. 모호성을 제거하려면 *PyErr_Occurred()*를 사용하십시오.

`unsigned long long PyLong_AsUnsignedLongLong (PyObject *pylong)`

*pylong*의 C unsigned long long 표현을 반환합니다. *pylong*은 *PyLongObject*의 인스턴스여야 합니다.

*pylong*의 값이 unsigned long long의 범위를 벗어나면 *OverflowError*를 발생시킵니다.

에러 시 (unsigned long long)-1을 반환합니다. 모호성을 제거하려면 *PyErr_Occurred()*를 사용하십시오.

버전 3.1에서 변경: 음의 *pylong*은 이제 *TypeError*가 아니라 *OverflowError*를 발생시킵니다.

`unsigned long PyLong_AsUnsignedLongMask (PyObject *obj)`

*obj*의 C unsigned long 표현을 반환합니다. *obj*가 *PyLongObject*의 인스턴스가 아니면, (있다면) 먼저 *__index__()* 나 *__int__()* 메서드를 호출하여 *PyLongObject*로 변환합니다.

*obj*의 값이 unsigned long의 범위를 벗어나면, 그 값의 모듈로 `ULONG_MAX + 1` 환원을 반환합니다.

에러 시 (unsigned long)-1을 반환합니다. 모호성을 제거하려면 *PyErr_Occurred()*를 사용하십시오.

버전 3.8에서 변경: 사용할 수 있으면 *__index__()*를 사용합니다.

버전 3.8부터 폐지: *__int__()* 사용은 폐지되었습니다.

`unsigned long long PyLong_AsUnsignedLongLongMask (PyObject *obj)`

*obj*의 C unsigned long long 표현을 반환합니다. *obj*가 *PyLongObject*의 인스턴스가 아니면, (있다면) 먼저 *__index__()* 나 *__int__()* 메서드를 호출하여 *PyLongObject*로 변환합니다.

*obj*의 값이 unsigned long long의 범위를 벗어나면, 그 값의 모듈로 `PY_ULLONG_MAX + 1` 환원을 반환합니다.

에러 시 (unsigned long long)-1을 반환합니다. 모호성을 제거하려면 *PyErr_Occurred()*를 사용하십시오.

버전 3.8에서 변경: 사용할 수 있으면 *__index__()*를 사용합니다.

버전 3.8부터 폐지: *__int__()* 사용은 폐지되었습니다.

`double PyLong_AsDouble (PyObject *pylong)`

*pylong*의 C double 표현을 반환합니다. *pylong*은 *PyLongObject*의 인스턴스여야 합니다.

*pylong*의 값이 double의 범위를 벗어나면 *OverflowError*를 발생시킵니다.

에러 시 -1.0을 반환합니다. 모호성을 제거하려면 *PyErr_Occurred()*를 사용하십시오.

`void* PyLong_AsVoidPtr (PyObject *pylong)`

파이썬 정수 *pylong*을 C void 포인터로 변환합니다. *pylong*을 변환할 수 없으면, *OverflowError*가 발생합니다. 이것은 *PyLong_FromVoidPtr()*로 만들어진 값에 대해서만 사용할 수 있는 void 포인터를 생성하는 것이 보장됩니다.

에러 시 NULL을 반환합니다. 모호성을 제거하려면 *PyErr_Occurred()*를 사용하십시오.

8.2.2 불리언 객체

파이썬에서 불리언은 정수의 서브 클래스로 구현됩니다. *Py_False*와 *Py_True*라는 두 개의 부울만 있습니다. 따라서 일반적인 생성 및 삭제 함수는 부울에 적용되지 않습니다. 그러나 다음 매크로를 사용할 수 있습니다.

`int PyBool_Check (PyObject *o)`

*o*가 *PyBool_Type* 형이면 참을 돌려줍니다.

***PyObject** Py_False**

파이썬 False 객체. 이 객체는 메서드가 없습니다. 참조 횟수와 관련해서는 다른 객체와 마찬가지로 처리해야 합니다.

***PyObject** Py_True**

파이썬 True 객체. 이 객체는 메서드가 없습니다. 참조 횟수와 관련해서는 다른 객체와 마찬가지로 처리해야 합니다.

Py_RETURN_FALSE

함수에서 Py_False를 반환하고, 참조 횟수를 적절하게 증가시킵니다.

Py_RETURN_TRUE

함수에서 Py_True를 반환하고, 참조 횟수를 적절하게 증가시킵니다.

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

Return value: New reference. v의 논리값에 따라 Py_True 나 Py_False에 대한 새 참조를 반환합니다.

8.2.3 부동 소수점 객체

PyFloatObject

이 *PyObject*의 서브 형은 파이썬 부동 소수점 객체를 나타냅니다.

***PyTypeObject* PyFloat_Type**

이 *PyTypeObject* 인스턴스는 파이썬 부동 소수점 형을 나타냅니다. 이것은 파이썬 계층에서 float와 같은 객체입니다.

int PyFloat_Check (*PyObject* *p)

인자가 *PyFloatObject* 나 *PyFloatObject*의 서브 형이면 참을 반환합니다.

int PyFloat_CheckExact (*PyObject* *p)

인자가 *PyFloatObject*이지만 *PyFloatObject*의 서브 형은 아니면 참을 반환합니다.

***PyObject** PyFloat_FromString (*PyObject* *str)**

Return value: New reference. str의 문자열 값을 기반으로 *PyFloatObject* 객체를 만들거나, 실패하면 NULL.

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

Return value: New reference. v로부터 *PyFloatObject* 객체를 만들거나, 실패하면 NULL.

double PyFloat_AsDouble (*PyObject* *pyfloat)

pyfloat의 내용의 C double 표현을 반환합니다. pyfloat가 파이썬 부동 소수점 객체가 아니지만 `__float__()` 메서드가 있으면, pyfloat를 float로 변환하기 위해 이 메서드가 먼저 호출됩니다. `__float__()`가 정의되지 않았으면 `__index__()`로 대체합니다. 이 메서드는 실패하면 -1.0을 반환하므로, `PyErr_Occurred()`를 호출하여 에러를 확인해야 합니다.

버전 3.8에서 변경: 사용할 수 있으면 `__index__()`를 사용합니다.

double PyFloat_AS_DOUBLE (*PyObject* *pyfloat)

에러 검사 없이 pyfloat의 내용의 C double 표현을 반환합니다.

***PyObject** PyFloat_GetInfo (void)**

Return value: New reference. float의 정밀도, 최솟값, 최댓값에 관한 정보를 포함한 structseq 인스턴스를 돌려줍니다. 헤더 파일 float.h를 감싸는 얇은 래퍼입니다.

double PyFloat_GetMax ()

최대 표현 가능한 유한 float DBL_MAX를 C double로 반환합니다.

double PyFloat_GetMin ()

최소 정규화된 (normalized) 양의 float DBL_MIN를 C double로 반환합니다.

`int PyFloat_ClearFreeList()`

float 자유 목록 (free list)을 비웁니다. 해제할 수 없는 항목의 수를 반환합니다.

8.2.4 복소수 객체

파이썬의 복소수 객체는 C API에서 볼 때 두 개의 다른 형으로 구현됩니다: 하나는 파이썬 프로그램에 노출된 파이썬 객체이고, 다른 하나는 실제 복소수 값을 나타내는 C 구조체입니다. API는 두 가지 모두도 작업할 수 있는 함수를 제공합니다.

C 구조체로서의 복소수

매개 변수로 이러한 구조체를 받아들이고 결과로 반환하는 함수는 포인터를 통해 역참조하기보다는 값으로 다룹니다. 이는 API 전체에서 일관됩니다.

`Py_complex`

파이썬 복소수 객체의 값 부분에 해당하는 C 구조체. 복소수 객체를 다루는 대부분 함수는 이 형의 구조체를 입력 또는 출력값으로 적절하게 사용합니다. 다음과 같이 정의됩니다:

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

`Py_complex Py_c_sum(Py_complex left, Py_complex right)`

C `Py_complex` 표현을 사용하여 두 복소수의 합을 반환합니다.

`Py_complex Py_c_diff(Py_complex left, Py_complex right)`

C `Py_complex` 표현을 사용하여 두 복소수의 차이를 반환합니다.

`Py_complex Py_c_neg(Py_complex complex)`

C `Py_complex` 표현을 사용하여 복소수 `complex`의 음의 값을 반환합니다.

`Py_complex Py_c_prod(Py_complex left, Py_complex right)`

C `Py_complex` 표현을 사용하여 두 복소수의 곱을 반환합니다.

`Py_complex Py_c_quot(Py_complex dividend, Py_complex divisor)`

C `Py_complex` 표현을 사용하여 두 복소수의 몫을 반환합니다.

`divisor`가 null이면, 이 메서드는 0을 반환하고, `errno`를 EDOM으로 설정합니다.

`Py_complex Py_c_pow(Py_complex num, Py_complex exp)`

C `Py_complex` 표현을 사용하여 `num`의 `exp` 거듭제곱을 반환합니다.

`num`이 null이고 `exp`가 양의 실수가 아니면, 이 메서드는 0을 반환하고 `errno`를 EDOM으로 설정합니다.

파이썬 객체로서의 복소수

`PyComplexObject`

파이썬 복소수 객체를 나타내는 `PyObject`의 서브 형.

`PyObject PyComplex_Type`

이 `PyObject` 인스턴스는 파이썬 복소수 형을 나타냅니다. 파이썬 계층의 `complex`와 같은 객체입니다.

`int PyComplex_Check(PyObject *p)`

인자가 `PyComplexObject` 나 `PyComplexObject`의 서브 형이면 참을 반환합니다.

`int PyComplex_CheckExact (PyObject *p)`

인자가 `PyComplexObject`이지만, `PyComplexObject`의 서브 유형이 아니면 참을 반환합니다.

`PyObject* PyComplex_FromCComplex (Py_complex v)`

Return value: New reference. C `Py_complex` 값으로 새로운 파이썬 복소수 객체를 만듭니다.

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

Return value: New reference. `real` 및 `imag`로 새로운 `PyComplexObject` 객체를 반환합니다.

`double PyComplex_RealAsDouble (PyObject *op)`

`op`의 실수부를 C `double`로 반환합니다.

`double PyComplex_ImagAsDouble (PyObject *op)`

`op`의 허수부를 C `double`로 반환합니다.

`Py_complex PyComplex_AsCComplex (PyObject *op)`

복소수 `op`의 `Py_complex` 값을 반환합니다.

`op`가 파이썬 복소수 객체가 아니지만 `__complex__()` 메서드가 있으면, 이 메서드는 먼저 `op`를 파이썬 복소수 객체로 변환하도록 그 메서드를 호출합니다. `__complex__()`가 정의되지 않았으면 `__float__()`로 대체합니다. `__float__()`가 정의되지 않았으면 `__index__()`로 대체합니다. 실패하면, 이 메서드는 `-1.0`를 실숫값으로 반환합니다.

버전 3.8에서 변경: 사용할 수 있다면 `__index__()`를 사용합니다.

8.3 시퀀스 객체

시퀀스 객체에 대한 일반적인 연산은 이전 장에서 논의했습니다; 이 절에서는 파이썬 언어에 고유한 특정 종류의 시퀀스 객체를 다룹니다.

8.3.1 바이트열 객체

이 함수들은 바이트열 매개 변수가 필요할 때 바이트열이 아닌 매개 변수로 호출하면 `TypeError`를 발생시킵니다.

PyBytesObject

이 `PyObject`의 서브 형은 파이썬 바이트열 객체를 나타냅니다.

PyTypeObject **PyBytes_Type**

이 `PyTypeObject`의 인스턴스는 파이썬 바이트열 형을 나타냅니다; 파이썬 계층의 `bytes`와 같은 객체입니다.

`int PyBytes_Check (PyObject *o)`

객체 `o`가 바이트열 객체이거나 바이트열 형의 서브 형의 인스턴스면 참을 반환합니다.

`int PyBytes_CheckExact (PyObject *o)`

객체 `o`가 바이트열 객체이지만, 바이트열 형의 서브 형의 인스턴스는 아니면 참을 반환합니다.

`PyObject* PyBytes_FromString (const char *v)`

Return value: New reference. 성공하면 값으로 `v` 문자열의 복사본을 갖는 새 바이트열 객체를 반환하고, 실패하면 `NULL`을 반환합니다. 매개 변수 `v`는 `NULL`이 아니어야 합니다; 검사하지 않습니다.

`PyObject* PyBytes_FromStringAndSize (const char *v, Py_ssize_t len)`

Return value: New reference. 성공하면 값이 `v` 문자열의 복사본이고 길이가 `len`인 새 바이트열 객체를 반환하고, 실패하면 `NULL`을 반환합니다. `v`가 `NULL`이면, 바이트열 객체의 내용은 초기화되지 않습니다.

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

Return value: New reference. C printf()-스타일 format 문자열과 가변 개수의 인자를 받아서, 결과 파인된 바이트열 객체의 크기를 계산하고 그 안에 값이 포맷된 바이트열 객체를 반환합니다. 가변 인자는 C 형이어야 하며 format 문자열에 있는 포맷 문자들과 정확히 대응해야 합니다. 허용되는 포맷 문자는 다음과 같습니다:

포맷 문자	형	주석
%%	n/a	리터럴 % 문자.
%c	int	단일 바이트, C int로 표현됩니다.
%d	int	printf("%d")와 동등합니다. ¹
%u	unsigned int	printf("%u")와 동등합니다. ¹
%ld	long	printf("%ld")와 동등합니다. ¹
%lu	unsigned long	printf("%lu")와 동등합니다. ¹
%zd	Py_ssize_t	printf("%zd")와 동등합니다. ¹
%zu	size_t	printf("%zu")와 동등합니다. ¹
%i	int	printf("%i")와 동등합니다. ¹
%x	int	printf("%x")와 동등합니다. ¹
%s	const char*	널-종료 C 문자 배열.
%p	const void*	C 포인터의 16진수 표현. 플랫폼의 printf가 어떤 결과를 내는지에 상관없이 리터럴 0x로 시작함이 보장된다는 점을 제외하고는 거의 printf("%p")와 동등합니다.

인식할 수 없는 포맷 문자는 포맷 문자열의 나머지 부분이 모두 결과 객체에 그대로 복사되게 만들고, 추가 인자는 무시됩니다.

*PyObject** **PyBytes_FromFormatV** (const char *format, va_list vars)

Return value: New reference. 정확히 두 개의 인자를 취한다는 것을 제외하고는 *PyBytes_FromFormat()*과 같습니다.

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

Return value: New reference. 버퍼 프로토콜을 구현하는 객체 o의 바이트열 표현을 반환합니다.

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

바이트열 객체 o의 길이를 반환합니다.

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

에러 검사 없는 *PyBytes_Size()*의 매크로 형식.

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

o의 내용에 대한 포인터를 반환합니다. 포인터는 len(o) + 1 바이트로 구성된 o의 내부 버퍼를 가리킵니다. 버퍼의 마지막 바이트는 다른 널(null) 바이트가 있는지에 관계없이 항상 널입니다. 객체가 PyBytes_FromStringAndSize(NULL, size)를 사용하여 방금 만들어진 경우가 아니면 데이터를 수정해서는 안 됩니다. 할당을 해제해서는 안 됩니다. o가 바이트열 객체가 아니면, *PyBytes_AsString()*은 NULL을 반환하고 TypeError를 발생시킵니다.

char* **PyBytes_AS_STRING** (*PyObject* *string)

에러 검사 없는 *PyBytes_AsString()*의 매크로 형식.

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

출력 변수 buffer와 length로 객체 obj의 널-종료 내용을 반환합니다.

length가 NULL이면, 바이트열 객체는 내장된 널 바이트를 포함할 수 없습니다; 만약 그렇다면 함수는 -1을 반환하고 ValueError를 발생시킵니다.

buffer는 obj의 내부 버퍼를 가리키게 되는데, 끝에 추가 널 바이트가 포함됩니다 (length에는 포함되지 않습니다). 객체가 PyBytes_FromStringAndSize(NULL, size)를 사용하여 방금 만들어진 경우가

¹ 정수 지정자(d, u, ld, lu, zd, zu, i, x)에서: 0-변환 플래그는 정밀도를 지정해도 영향을 미칩니다.

아니면 데이터를 수정해서는 안 됩니다. 할당을 해제해서는 안 됩니다. *obj*가 바이트열 객체가 아니면 *PyBytes_AsStringAndSize()*는 -1을 반환하고 *TypeError*를 발생시킵니다.

버전 3.5에서 변경: 이전에는, 바이트열 객체에 널 바이트가 포함되어 있으면 *TypeError*가 발생했습니다.

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

*bytes*에 *newpart*의 내용을 덧붙인 새 바이트열 객체를 **bytes*에 만듭니다; 호출자가 새 참조를 소유합니다. *bytes*의 예전 값에 대한 참조를 훔칩니다. 새 객체가 만들어질 수 없으면, *bytes*에 대한 예전 참조는 여전히 버려지고 **bytes*의 값은 *NULL*로 설정됩니다; 적절한 예외가 설정됩니다.

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

*bytes*에 *newpart*의 내용을 덧붙인 새 바이트열 객체를 **bytes*에 만듭니다. 이 버전은 *newpart*의 참조 횟수를 감소시킵니다.

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

“불변”임에도 불구하고 바이트열 객체의 크기를 변경하는 방법. 완전히 새로운 바이트열 객체를 만들 때만 사용하십시오; *bytes*가 이미 코드의 다른 부분에 알려져 있을 수 있다면 사용하지 마십시오. 입력 바이트열 객체의 참조 횟수가 1이 아닐 때 이 함수를 호출하는 것은 에러입니다. 기존 바이트열 객체의 주소를 *lvalue*(내용을 기록할 수 있습니다)로 전달하고, 원하는 새 크기를 전달합니다. 성공하면, **bytes*는 크기가 변경된 바이트열 객체를 갖게 되고 0이 반환됩니다; **bytes*의 주소는 입력값과 다를 수 있습니다. 재할당이 실패하면, **bytes*에 있는 원래 바이트열 객체는 할당 해제되고, **bytes*가 *NULL*로 설정되고, *MemoryError*가 설정되며 -1이 반환됩니다.

8.3.2 바이트 배열 객체

PyByteArrayObject

이 *PyObject*의 서브 형은 파이썬 bytearray 객체를 나타냅니다.

PyTypeObject **PyByteArray_Type**

이 *PyTypeObject* 인스턴스는 파이썬 bytearray 형을 나타냅니다; 파이썬 계층의 bytearray와 같은 객체입니다.

형 검사 매크로

int **PyByteArray_Check** (*PyObject* **o*)

객체 *o*가 bytearray 객체이거나 bytearray 형의 서브 형 인스턴스면 참을 반환합니다.

int **PyByteArray_CheckExact** (*PyObject* **o*)

객체 *o*가 bytearray 객체이지만, bytearray 형의 서브 형 인스턴스는 아니면 참을 반환합니다.

직접 API 함수

*PyObject** **PyByteArray_FromObject** (*PyObject* **o*)

Return value: New reference. 버퍼 프로토콜을 구현하는 임의의 객체(*o*)로부터 써서 새로운 bytearray 객체를 돌려줍니다.

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

Return value: New reference. *string*과 그 길이(*len*)로부터 새로운 bytearray 객체를 만듭니다. 실패하면, *NULL*이 반환됩니다.

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

Return value: New reference. 바이트 배열 *a*와 *b*를 이어붙여 새로운 bytearray로 반환합니다.

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

NULL 포인터를 확인한 후 bytearray의 크기를 반환합니다.

`char* PyByteArray_AsString (PyObject *bytearray)`
 NULL 포인터를 확인한 후 `bytearray`의 내용을 `char` 배열로 반환합니다. 반환되는 배열에는 항상 여분의 널 바이트가 추가됩니다.

`int PyByteArray_Resize (PyObject *bytearray, Py_ssize_t len)`
`bytearray`의 내부 버퍼의 크기를 `len`으로 조정합니다.

매크로

이 매크로는 속도를 위해 안전을 희생하며 포인터를 확인하지 않습니다.

`char* PyByteArray_AS_STRING (PyObject *bytearray)`
`PyByteArray_AsString()`의 매크로 버전.

`Py_ssize_t PyByteArray_GET_SIZE (PyObject *bytearray)`
`PyByteArray_Size()`의 매크로 버전.

8.3.3 유니코드 객체와 코덱

유니코드 객체

파이썬 3.3에서 **PEP 393**을 구현한 이후, 유니코드 객체는 내부적으로 다양한 표현을 사용하여 전체 유니코드 문자 범위를 처리하면서 메모리 효율성을 유지합니다. 모든 코드 포인트가 128, 256 또는 65536 미만인 문자열에 대한 특별한 경우가 있습니다; 그렇지 않으면, 코드 포인트는 1114112 (전체 유니코드 범위) 미만이어야 합니다.

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

이전 API와 새 API 간의 전환으로 인해, 유니코드 객체는 만들어진 방법에 따라 내부적으로 두 가지 상태가 될 수 있습니다:

- “규범적(canonical)” 유니코드 객체는 폐지되지 않은 유니코드 API에 의해 만들어진 모든 객체입니다. 구현에서 허용하는 가장 효율적인 표현을 사용합니다.
- “레거시” 유니코드 객체는 폐지된 API 중 하나(일반적으로 `PyUnicode_FromUnicode()`)를 통해 만들어지고 `Py_UNICODE*` 표현만 포함합니다; 다른 API를 호출하기 전에 이들에 대해 `PyUnicode_READY()`를 호출해야 합니다.

참고: “레거시” 유니코드 객체는 폐지된 API와 함께 파이썬 3.12에서 제거됩니다. 그 이후로 모든 유니코드 객체는 “규범적”이 됩니다. 자세한 정보는 **PEP 623**을 참조하십시오.

유니코드 형

다음은 파이썬에서 유니코드 구현에 사용되는 기본 유니코드 객체 형입니다:

Py_UCS4

Py_UCS2

Py_UCS1

이 형들은 각각 32비트, 16비트 및 8비트의 문자를 포함하기에 충분한 부호 없는 정수 형을 위한 typedef입니다. 단일 유니코드 문자를 처리할 때는, `Py_UCS4`를 사용하십시오.

버전 3.3에 추가.

Py_UNICODE

이것은 플랫폼에 따라 16비트 형이나 32비트 형인 `wchar_t`의 typedef 입니다.

버전 3.3에서 변경: 이전 버전에서, 이것은 빌드 시 파이썬의 “내로우(narrow)”나 “와이드(wide)” 유니코드 버전 중 어느 것을 선택했는지에 따라 16비트 형이나 32비트 형이었습니다.

PyASCIIObject**PyCompactUnicodeObject****PyUnicodeObject**

이 *PyObject* 서브 형들은 파이썬 유니코드 객체를 나타냅니다. 거의 모든 경우에, 유니코드 객체를 처리하는 모든 API 함수가 *PyObject* 포인터를 취하고 반환하므로 직접 사용해서는 안 됩니다.

버전 3.3에 추가.

PyTypeObject **PyUnicode_Type**

이 *PyTypeObject* 인스턴스는 파이썬 유니코드 형을 나타냅니다. 파이썬 코드에 `str`로 노출됩니다.

다음 API는 실제로는 C 매크로이며 빠른 검사를 수행하고 유니코드 객체의 내부 읽기 전용 데이터에 액세스하는 데 사용할 수 있습니다:

`int PyUnicode_Check (PyObject *o)`

객체 *o*가 유니코드 객체이거나 유니코드 서브 형의 인스턴스이면 참을 반환합니다.

`int PyUnicode_CheckExact (PyObject *o)`

객체 *o*가 유니코드 객체이지만, 서브 형의 인스턴스가 아니면 참을 반환합니다.

`int PyUnicode_READY (PyObject *o)`

문자열 객체 *o*가 “규범적(canonical)” 표현인지 확인합니다. 이것은 아래에 설명된 액세스 매크로를 사용하기 전에 필요합니다.

성공 시 0을 반환하고, 실패 시 예외를 설정하면서 -1을 반환하는데, 특히 메모리 할당이 실패하면 발생합니다.

버전 3.3에 추가.

Deprecated since version 3.10, will be removed in version 3.12: 이 API는 *PyUnicode_FromUnicode()*와 함께 제거됩니다.

`Py_ssize_t PyUnicode_GET_LENGTH (PyObject *o)`

유니코드 문자열의 길이를 코드 포인트로 반환합니다. *o*는 “규범적(canonical)” 표현의 유니코드 객체여야 합니다 (검사하지 않습니다).

버전 3.3에 추가.

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

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

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

직접 문자 액세스를 위해 UCS1, UCS2 또는 UCS4 정수 형으로 캐스트 된 규범적(canonical) 표현에 대한 포인터를 반환합니다. 규범적(canonical) 표현이 올바른 문자 크기인지 검사하지 않습니다; *PyUnicode_KIND()*를 사용하여 올바른 매크로를 선택하십시오. 이것을 액세스하기 전에 *PyUnicode_READY()*가 호출되었어야 합니다.

버전 3.3에 추가.

PyUnicode_WCHAR_KIND

PyUnicode_1BYTE_KIND

PyUnicode_2BYTE_KIND

PyUnicode_4BYTE_KIND

PyUnicode_KIND() 매크로의 값을 반환합니다.

버전 3.3에 추가.

Deprecated since version 3.10, will be removed in version 3.12: `PyUnicode_WCHAR_KIND`는 폐지되었습니다.

`int PyUnicode_KIND (PyObject *o)`

이 유니코드 객체가 데이터를 저장하는 데 사용하는 문자 당 바이트 수를 나타내는 `PyUnicode` 종류 상수 (위를 참조하십시오) 중 하나를 반환합니다. *o*는 “규범적(canonical)” 표현의 유니코드 객체여야 합니다 (검사하지 않습니다).

버전 3.3에 추가.

`void* PyUnicode_DATA (PyObject *o)`

원시 유니코드 버퍼에 대한 `void` 포인터를 반환합니다. *o*는 “규범적(canonical)” 표현의 유니코드 객체여야 합니다 (검사하지 않습니다).

버전 3.3에 추가.

`void PyUnicode_WRITE (int kind, void *data, Py_ssize_t index, Py_UCS4 value)`

규범적(canonical) 표현 `data(PyUnicode_DATA())`로 얻은 대로)에 씁니다. 이 매크로는 온전성 검사(sanity checks)를 수행하지 않으며 루프에서 사용하기 위한 것입니다. 호출자는 다른 매크로 호출에서 얻은 대로 *kind* 값과 *data* 포인터를 캐시 해야 합니다. *index*는 문자열의 인덱스(0에서 시작합니다)이고 *value*는 해당 위치에 기록되어야 하는 새 코드 포인트 값입니다.

버전 3.3에 추가.

`Py_UCS4 PyUnicode_READ (int kind, void *data, Py_ssize_t index)`

규범적(canonical) 표현 `data(PyUnicode_DATA())`로 얻은 대로)에서 코드 포인트를 읽습니다. 검사나 준비(ready) 호출이 수행되지 않습니다.

버전 3.3에 추가.

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

“규범적(canonical)” 표현이어야 하는, 유니코드 객체 *o*에서 문자를 읽습니다. 여러 연속 읽기를 수행한다면 `PyUnicode_READ()`보다 효율성이 떨어집니다.

버전 3.3에 추가.

`PyUnicode_MAX_CHAR_VALUE (o)`

“규범적(canonical)” 표현이어야 하는, *o*를 기반으로 다른 문자열을 만드는 데 적합한 최대 코드 포인트를 반환합니다. 이것은 항상 근사치이지만 문자열을 이터레이트 하는 것보다 효율적입니다.

버전 3.3에 추가.

`int PyUnicode_ClearFreeList ()`

Clear the free list. Return the total number of freed items.

`Py_ssize_t PyUnicode_GET_SIZE (PyObject *o)`

폐지된 `Py_UNICODE` 표현의 크기를 코드 단위로 반환합니다 (서로게이트 쌍을 2단위로 포함합니다). *o*는 유니코드 객체여야 합니다 (검사하지 않습니다).

Deprecated since version 3.3, will be removed in version 3.12: 이전 스타일 유니코드 API의 일부입니다, `PyUnicode_GET_LENGTH()`를 사용하여 마이그레이션 하십시오.

`Py_ssize_t PyUnicode_GET_DATA_SIZE (PyObject *o)`

폐지된 `Py_UNICODE` 표현의 크기를 바이트 단위로 반환합니다. *o*는 유니코드 객체여야 합니다 (검사하지 않습니다).

Deprecated since version 3.3, will be removed in version 3.12: 이전 스타일 유니코드 API의 일부입니다, `PyUnicode_GET_LENGTH()`를 사용하여 마이그레이션 하십시오.

`Py_UNICODE* PyUnicode_AS_UNICODE (PyObject *o)`

`const char* PyUnicode_AS_DATA (PyObject *o)`

객체의 `Py_UNICODE` 표현에 대한 포인터를 반환합니다. 반환된 버퍼는 항상 추가 널 코드 포인트로 끝납니다. 또한 내장된 널 코드 포인트를 포함할 수 있는데, 대부분의 C 함수에서 사용될 때 문자열이

잘리도록 합니다. `AS_DATA` 형식은 포인터를 `const char *`로 캐스트 합니다. `o` 인자는 유니코드 객체여야 합니다 (검사하지 않습니다).

버전 3.3에서 변경: 이 매크로는 이제 비효율적이고 – 많은 경우에 `Py_UNICODE` 표현이 존재하지 않고 만들어야 해서 – 실패할 수 있습니다 (예외 설정과 함께 `NULL`을 반환합니다). 새 `PyUnicode_nBYTE_DATA()` 매크로를 사용하거나 `PyUnicode_WRITE()` 나 `PyUnicode_READ()`를 사용하도록 코드를 이식하십시오.

Deprecated since version 3.3, will be removed in version 3.12: 이전 스타일 유니코드 API의 일부입니다. `PyUnicode_nBYTE_DATA()` 매크로 계열을 사용하도록 마이그레이션 하십시오.

유니코드 문자 속성

유니코드는 다양한 문자 속성을 제공합니다. 가장 자주 필요한 것은 파이썬 구성에 따라 C 함수에 매핑되는 이러한 매크로를 통해 사용할 수 있습니다.

`int Py_UNICODE_ISSPACE(Py_UNICODE ch)`
`ch`가 공백 문자인지에 따라 1이나 0을 반환합니다.

`int Py_UNICODE_ISLOWER(Py_UNICODE ch)`
`ch`가 소문자인지에 따라 1이나 0을 반환합니다.

`int Py_UNICODE_ISUPPER(Py_UNICODE ch)`
`ch`가 대문자인지에 따라 1이나 0을 반환합니다.

`int Py_UNICODE_ISTITLE(Py_UNICODE ch)`
`ch`가 제목 케이스 문자인지에 따라 1이나 0을 반환합니다.

`int Py_UNICODE_ISLINEBREAK(Py_UNICODE ch)`
`ch`가 줄 바꿈 문자인지에 따라 1이나 0을 반환합니다.

`int Py_UNICODE_ISDECIMAL(Py_UNICODE ch)`
`ch`가 10진수 문자인지에 따라 1이나 0을 반환합니다.

`int Py_UNICODE_ISDIGIT(Py_UNICODE ch)`
`ch`가 디짓(digit) 문자인지에 따라 1이나 0을 반환합니다.

`int Py_UNICODE_ISNUMERIC(Py_UNICODE ch)`
`ch`가 숫자(numeric) 문자인지에 따라 1이나 0을 반환합니다.

`int Py_UNICODE_ISALPHA(Py_UNICODE ch)`
`ch`가 알파벳 문자인지에 따라 1이나 0을 반환합니다.

`int Py_UNICODE_ISALNUM(Py_UNICODE ch)`
`ch`가 영숫자 문자인지에 따라 1이나 0을 반환합니다.

`int Py_UNICODE_ISPRINTABLE(Py_UNICODE ch)`
`ch`가 인쇄 가능한 문자인지에 따라 1이나 0을 반환합니다. 인쇄할 수 없는 문자는, 인쇄 가능한 것으로 간주하는 ASCII 스페이스(0x20)를 제외하고, 유니코드 문자 데이터베이스에서 “Other”나 “Separator”로 정의된 문자입니다. (이 문맥에서 인쇄 가능한 문자는 `repr()`이 문자열에 대해 호출될 때 이스케이프되지 않아야 하는 문자임에 유의하십시오. `sys.stdout`이나 `sys.stderr`에 기록된 문자열의 처리와 관련이 없습니다.)

다음 API는 빠른 직접 문자 변환에 사용할 수 있습니다:

`Py_UNICODE Py_UNICODE_TOLOWER(Py_UNICODE ch)`
소문자로 변환된 문자 `ch`를 반환합니다.

버전 3.3부터 폐지: 이 함수는 간단한 케이스 매핑을 사용합니다.

Py_UNICODE **Py_UNICODE_TOUPPER** (*Py_UNICODE* *ch*)

대문자로 변환된 문자 *ch*를 반환합니다.

버전 3.3부터 폐지: 이 함수는 간단한 케이스 매핑을 사용합니다.

Py_UNICODE **Py_UNICODE_TOTITLE** (*Py_UNICODE* *ch*)

제목 케이스로 변환된 문자 *ch*를 반환합니다.

버전 3.3부터 폐지: 이 함수는 간단한 케이스 매핑을 사용합니다.

int **Py_UNICODE_TODECIMAL** (*Py_UNICODE* *ch*)

10진 양의 정수로 변환된 문자 *ch*를 반환합니다. 이것이 불가능하면 -1을 반환합니다. 이 매크로는 예외를 발생시키지 않습니다.

int **Py_UNICODE_TODIGIT** (*Py_UNICODE* *ch*)

한 자리 정수로 변환된 문자 *ch*를 반환합니다. 이것이 불가능하면 -1을 반환합니다. 이 매크로는 예외를 발생시키지 않습니다.

double **Py_UNICODE_TONUMERIC** (*Py_UNICODE* *ch*)

double로 변환된 문자 *ch*를 반환합니다. 이것이 불가능하면 -1.0을 반환합니다. 이 매크로는 예외를 발생시키지 않습니다.

다음 API를 사용하여 서로게이트를 다룰 수 있습니다:

Py_UNICODE_IS_SURROGATE (*ch*)

*ch*가 서로게이트인지 확인합니다 (0xD800 <= *ch* <= 0xDFFF).

Py_UNICODE_IS_HIGH_SURROGATE (*ch*)

*ch*가 상위 서로게이트인지 확인합니다 (0xD800 <= *ch* <= 0xDBFF).

Py_UNICODE_IS_LOW_SURROGATE (*ch*)

*ch*가 하위 서로게이트인지 확인합니다 (0xDC00 <= *ch* <= 0xDFFF).

Py_UNICODE_JOIN_SURROGATES (*high*, *low*)

두 서로게이트 문자를 결합하고 단일 Py_UCS4 값을 반환합니다. *high*와 *low*는 각각 서로게이트 쌍의 선행과 후행 서로게이트입니다.

유니코드 문자열 생성과 액세스

유니코드 객체를 만들고 기본 시퀀스 속성에 액세스하려면 다음 API를 사용하십시오:

*PyObject** **PyUnicode_New** (*Py_ssize_t* *size*, *Py_UCS4* *maxchar*)

Return value: *New reference.* 새 유니코드 객체를 만듭니다. *maxchar*은 문자열에 배치할 실제 최대 코드 포인트여야 합니다. 근삿값으로, 127, 255, 65535, 1114111 시퀀스에서 가장 가까운 값으로 올림 할 수 있습니다.

이것은 새 유니코드 객체를 할당하는 데 권장되는 방법입니다. 이 함수를 사용하여 만든 객체는 크기를 조정할 수 없습니다.

버전 3.3에 추가.

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

Return value: *New reference.* 주어진 *kind*(가능한 값은 *PyUnicode_KIND()*에 의해 반환된 *PyUnicode_1BYTE_KIND* 등입니다)로 새로운 유니코드 객체를 만듭니다. *buffer*는 *kind*에 따라 문자 당 1, 2 또는 4바이트의 *size* 단위의 배열을 가리켜야 합니다.

버전 3.3에 추가.

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

Return value: *New reference.* char 버퍼 *u*에서 유니코드 객체를 만듭니다. 바이트는 UTF-8로 인코딩된

것으로 해석됩니다. 버퍼는 새 객체에 복사됩니다. 버퍼가 NULL이 아니면, 반환 값은 공유 객체일 수 있습니다, 즉, 데이터 수정이 허용되지 않습니다.

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. UTF-8로 인코딩된 널-종료 char 버퍼 *u*에서 유니코드 객체를 만듭니다.

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

Return value: New reference. C `printf()`-스타일 *format* 문자열과 가변 개수의 인자를 취해서, 결과 파이썬 유니코드 문자열의 크기를 계산하고 포맷된 값이 들어간 문자열을 반환합니다. 변수 인자는 C 형이어야 하며 *format* ASCII 인코딩된 문자열의 포맷 문자와 정확히 일치해야 합니다. 다음 포맷 문자가 허용됩니다:

포맷 문자	형	주석
%%	<i>n/a</i>	리터럴 % 문자.
%c	int	C int로 표현된, 단일 문자.
%d	int	<code>printf("%d")</code> 와 동등합니다. ¹
%u	unsigned int	<code>printf("%u")</code> 와 동등합니다. ¹
%ld	long	<code>printf("%ld")</code> 와 동등합니다. ¹
%li	long	<code>printf("%li")</code> 와 동등합니다. ¹
%lu	unsigned long	<code>printf("%lu")</code> 와 동등합니다. ¹
%lld	long long	<code>printf("%lld")</code> 와 동등합니다. ¹
%lli	long long	<code>printf("%lli")</code> 와 동등합니다. ¹
%llu	unsigned long long	<code>printf("%llu")</code> 와 동등합니다. ¹
%zd	Py_ssize_t	<code>printf("%zd")</code> 와 동등합니다. ¹
%zi	Py_ssize_t	<code>printf("%zi")</code> 와 동등합니다. ¹
%zu	size_t	<code>printf("%zu")</code> 와 동등합니다. ¹
%i	int	<code>printf("%i")</code> 와 동등합니다. ¹
%x	int	<code>printf("%x")</code> 와 동등합니다. ¹
%s	const char*	널-종료 C 문자 배열.
%p	const void*	C 포인터의 16진수 표현. 플랫폼의 <code>printf</code> 가 산출하는 내용과 관계없이 리터럴 0x로 시작하는 것이 보장된다는 점을 제외하면 거의 <code>printf("%p")</code> 와 동등합니다.
%A	PyObject*	<code>ascii()</code> 를 호출한 결과.
%U	PyObject*	유니코드 객체.
%V	PyObject*, const char*	유니코드 객체(NULL일 수 있습니다)와 두 번째 매개 변수로서 널-종료 C 문자 배열 (첫 번째 매개 변수가 NULL이면 사용됩니다).
%S	PyObject*	<code>PyObject_Str()</code> 을 호출한 결과.
%R	PyObject*	<code>PyObject_Repr()</code> 을 호출한 결과.

인식할 수 없는 포맷 문자는 나머지 포맷 문자열이 모두 결과 문자열에 그대로 복사되고, 추가 인자는 버려지도록 합니다.

참고: 너비 포매터 단위는 바이트가 아닌 문자 수입니다. 정밀도 포매터 단위는 "%s"와 "%V"의 경우는 바이트 수이고 (PyObject* 인자가 NULL이면), "%A", "%U", "%S", "%R" 및 "%V"의 경우 문자 수입니다 (PyObject* 인자가 NULL이 아니면).

버전 3.2에서 변경: "%lld"와 "%llu"에 대한 지원이 추가되었습니다.

¹ 정수 지정자 (d, u, ld, lu, lld, lli, llu, zd, zi, zu, i, x)의 경우: 0-변환 플래그는 정밀도가 제공되는 경우에도 적용됩니다.

버전 3.3에서 변경: "%li", "%lli" 및 "%zi"에 대한 지원이 추가되었습니다.

버전 3.4에서 변경: "%s", "%A", "%U", "%V", "%S", "%R"에 대한 너비와 정밀도 포맷터 지원이 추가되었습니다.

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

Return value: New reference. 정확히 두 개의 인자를 취한다는 점을 제외하면 *PyUnicode_FromFormat()*과 동일합니다.

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

Return value: New reference. 인코딩된 객체 *obj*를 유니코드 객체로 디코딩합니다.

bytes, bytearray 및 기타 바이트열 객체는 주어진 *encoding*에 따라 *errors*로 정의한 에러 처리를 사용하여 디코딩됩니다. 둘 다 NULL이 될 수 있고, 이 경우 인터페이스는 기본값을 사용합니다 (자세한 내용은 내장 코텍를 참조하십시오).

유니코드 객체를 포함한 다른 모든 객체는 *TypeError*가 설정되도록 합니다.

API는 에러가 있으면 NULL을 반환합니다. 호출자는 반환된 객체의 참조 횟수를 감소시킬 책임이 있습니다.

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

유니코드 객체의 길이를 코드 포인트로 반환합니다.

버전 3.3에 추가.

Py_ssize_t **PyUnicode_CopyCharacters** (*PyObject* *to, Py_ssize_t to_start, *PyObject* *from, Py_ssize_t from_start, Py_ssize_t how_many)

한 유니코드 객체에서 다른 객체로 문자를 복사합니다. 이 함수는 필요하면 문자 변환을 수행하고 가능하면 *memcpy()*로 폴백합니다. 에러 시 -1을 반환하고 예외를 설정합니다, 그렇지 않으면 복사된 문자 수를 반환합니다.

버전 3.3에 추가.

Py_ssize_t **PyUnicode_Fill** (*PyObject* *unicode, Py_ssize_t start, Py_ssize_t length, *Py_UCS4* fill_char)

문자로 문자열을 채웁니다: *fill_char*을 *unicode[start:start+length]*에 씁니다.

*fill_char*이 문자열 최대 문자보다 크거나, 문자열에 둘 이상의 참조가 있으면 실패합니다.

기록된 문자 수를 반환하거나, 에러 시 -1을 반환하고 예외를 발생시킵니다.

버전 3.3에 추가.

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

문자열에 문자를 씁니다. 문자열은 *PyUnicode_New()*를 통해 만들어야 합니다. 유니코드 문자열은 불변이므로, 문자열을 공유하거나 아직 해시 하지 않아야 합니다.

이 함수는 *unicode*가 유니코드 객체인지, 인덱스가 범위를 벗어났는지, 객체가 안전하게 수정될 수 있는지 (즉, 참조 횟수가 1인지) 확인합니다.

버전 3.3에 추가.

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

문자열에서 문자를 읽습니다. 이 함수는 매크로 버전 *PyUnicode_READ_CHAR()*와 달리 *unicode*가 유니코드 객체이고 인덱스가 범위를 벗어났는지 확인합니다.

버전 3.3에 추가.

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

Return value: New reference. 문자 인덱스 *start*(포함합니다)에서 문자 인덱스 *end*(제외합니다)까지 *str*의 하위 문자열을 반환합니다. 음수 인덱스는 지원되지 않습니다.

버전 3.3에 추가.

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

*copy_null*이 설정되면, 널 문자를 포함하여 문자열 *u*를 UCS4 버퍼에 복사합니다. 예러 시 NULL을 반환하고 예외를 설정합니다 (특히, *buflen*이 *u*의 길이보다 작으면 `SystemError`). 성공하면 *buffer*가 반환됩니다.

버전 3.3에 추가.

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

문자열 *u*를 `PyMem_Malloc()`을 사용하여 할당된 새 UCS4 버퍼에 복사합니다. 이것이 실패하면, NULL이 반환되고 `MemoryError`가 설정됩니다. 반환된 버퍼에는 항상 추가 널 코드 포인트가 있습니다.

버전 3.3에 추가.

폐지된 Py_UNICODE API

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

이 API 함수들은 **PEP 393** 구현에 의해 폐지되었습니다. 파이썬 3.x에서 제거되지 않기 때문에, 확장 모듈은 계속해서 사용할 수 있지만, 이제 그 사용으로 인해 성능과 메모리 문제가 있을 수 있음을 인식해야 합니다.

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

Return value: *New reference.* 주어진 크기(*size*)의 `Py_UNICODE` 버퍼 *u*에서 유니코드 객체를 만듭니다. *u*는 NULL일 수 있으며, 이럴 때는 내용이 정의되지 않습니다. 필요한 데이터를 채우는 것은 사용자의 책임입니다. 버퍼가 새 객체에 복사됩니다.

버퍼가 NULL이 아니면, 반환 값은 공유 객체일 수 있습니다. 따라서, 결과 유니코드 객체의 수정은 *u*가 NULL일 때만 허용됩니다.

버퍼가 NULL이면, `PyUnicode_KIND()`와 같은 액세스 매크로를 사용하기 전에 문자열 내용이 채워지면 `PyUnicode_READY()`를 호출해야 합니다.

Deprecated since version 3.3, will be removed in version 3.12: 이전 스타일 유니코드 API의 일부입니다. `PyUnicode_FromKindAndData()`, `PyUnicode_FromWideChar()` 또는 `PyUnicode_New()`를 사용하여 마이그레이션 하십시오.

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

유니코드 객체의 내부 `Py_UNICODE` 버퍼에 대한 읽기 전용 포인터를 반환하거나, 예러 시 NULL을 반환합니다. 아직 사용할 수 없으면 객체의 `Py_UNICODE*` 표현을 만듭니다. 버퍼는 항상 여분의 널 코드 포인트로 종료됩니다. 결과 `Py_UNICODE` 문자열에는 내장된 널 코드 포인트도 포함될 수 있으며, 이때는 대부분의 C 함수에서 사용될 때 문자열이 잘림에 유의하십시오.

Deprecated since version 3.3, will be removed in version 3.12: 이전 스타일 유니코드 API의 일부입니다. `PyUnicode_AsUCS4()`, `PyUnicode_AsWideChar()`, `PyUnicode_ReadChar()` 또는 유사한 새 API를 사용하여 마이그레이션 하십시오.

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.* 주어진 *size*의 `Py_UNICODE` 버퍼에 있는 모든 10진 숫자들을 10진 값에 따라 ASCII 숫자 0-9로 대체하여 유니코드 객체를 만듭니다. 예외가 발생하면 NULL을 반환합니다.

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*)

`PyUnicode_AsUnicode()`와 비슷하지만, `Py_UNICODE()` 배열 길이(추가 널 종료 제외)를 *size*에 저장하기도 합니다. 결과 `Py_UNICODE*` 문자열에는 내장된 널 코드 포인트가 포함될 수 있으며, 이때는 대부분의 C 함수에서 사용될 때 문자열이 잘림에 유의하십시오.

버전 3.3에 추가.

Deprecated since version 3.3, will be removed in version 3.12: 이전 스타일 유니코드 API의 일부입니다. `PyUnicode_AsUCS4()`, `PyUnicode_AsWideChar()`, `PyUnicode_ReadChar()` 또는 유사한 새 API를 사용하여 마이그레이션 하십시오.

Py_UNICODE* PyUnicode_AsUnicodeCopy (PyObject *unicode)

널 코드 포인트로 끝나는 유니코드 문자열의 복사본을 만듭니다. 메모리 할당 실패 시 NULL을 반환하고 `MemoryError` 예외를 발생시킵니다, 그렇지 않으면 새로 할당된 버퍼를 반환합니다 (버퍼를 해제하려면 `PyMem_Free()`를 사용하십시오). 결과 `Py_UNICODE*` 문자열에는 내장된 널 코드 포인트가 포함될 수 있으며, 이때는 대부분의 C 함수에서 사용될 때 문자열이 잘림에 유의하십시오.

버전 3.2에 추가.

`PyUnicode_AsUCS4Copy()` 나 유사한 새 API를 사용하여 마이그레이션 하십시오.

Py_ssize_t PyUnicode_GetSize (PyObject *unicode)

패지된 `Py_UNICODE` 표현의 크기를 코드 단위로 반환합니다 (서로게이트 쌍을 2단위로 포함합니다).

Deprecated since version 3.3, will be removed in version 3.12: 이전 스타일 유니코드 API의 일부입니다, `PyUnicode_GET_LENGTH()`를 사용하여 마이그레이션 하십시오.

PyObject* PyUnicode_FromObject (PyObject *obj)

Return value: New reference. 필요하다면 유니코드 서브 형의 인스턴스를 새로운 진짜 유니코드 객체에 복사합니다. `obj`가 이미 (서브 형이 아닌) 진짜 유니코드 객체이면, 참조 횟수를 증가시키고 참조를 반환합니다.

유니코드나 이의 서브 형 이외의 객체는 `TypeError`를 발생시킵니다.

로케일 인코딩

현재 로케일 인코딩을 사용하여 운영 체제에서 온 텍스트를 디코딩 할 수 있습니다.

PyObject* PyUnicode_DecodeLocaleAndSize (const char *str, Py_ssize_t len, const char *errors)

Return value: New reference. 안드로이드와 VxWorks의 UTF-8이나 다른 플랫폼의 현재 로케일 인코딩의 문자열을 디코딩합니다. 지원되는 에러 처리기는 "strict"와 "surrogateescape"(PEP 383)입니다. 디코더는 `errors`가 NULL이면 "strict" 에러 처리기를 사용합니다. `str`은 널 문자로 끝나야 하지만 널 문자를 포함할 수 없습니다.

`Py_FileSystemDefaultEncoding`(파이썬 시작 시 읽은 로케일 인코딩)에서 문자열을 디코딩하려면 `PyUnicode_DecodeFSDefaultAndSize()`를 사용하십시오.

이 함수는 파이썬 UTF-8 모드를 무시합니다.

더 보기:

`Py_DecodeLocale()` 함수.

버전 3.3에 추가.

버전 3.7에서 변경: 이 함수는 이제 안드로이드를 제외하고 `surrogateescape` 에러 처리기에 현재 로케일 인코딩도 사용합니다. 이전에는, `Py_DecodeLocale()`이 `surrogateescape`에 사용되었고, 현재 로케일 인코딩은 `strict`에 사용되었습니다.

PyObject* PyUnicode_DecodeLocale (const char *str, const char *errors)

Return value: New reference. `PyUnicode_DecodeLocaleAndSize()`와 유사하지만, `strlen()`을 사용하여 문자열 길이를 계산합니다.

버전 3.3에 추가.

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

Return value: New reference. 유니코드 객체를 안드로이드와 VxWorks에서 UTF-8로 인코딩하거나, 다른 플랫폼에서 현재 로케일 인코딩으로 인코딩합니다. 지원되는 에러 처리기는 "strict"와

"surrogateescape"(PEP 383)입니다. 인코더는 *errors*가 NULL이면 "strict" 에러 처리기를 사용합니다. bytes 객체를 반환합니다. *unicode*는 내장된 널 문자를 포함할 수 없습니다.

문자열을 `Py_FileSystemDefaultEncoding`(파이썬 시작 시 읽은 로케일 인코딩)으로 인코딩하려면 `PyUnicode_EncodeFSDefault()`를 사용하십시오.

이 함수는 파이썬 UTF-8 모드를 무시합니다.

더 보기:

`Py_EncodeLocale()` 함수.

버전 3.3에 추가.

버전 3.7에서 변경: 이 함수는 이제 안드로이드를 제외하고 surrogateescape 에러 처리기에 현재 로케일 인코딩도 사용합니다. 이전에는 `Py_EncodeLocale()`이 surrogateescape에 사용되었고, 현재 로케일 인코딩은 strict에 사용되었습니다.

파일 시스템 인코딩

파일 이름과 기타 환경 문자열을 인코딩하고 디코딩하려면, `Py_FileSystemDefaultEncoding`을 인코딩으로 사용하고, `Py_FileSystemDefaultEncodeErrors`를 에러 처리기로 사용해야 합니다 (PEP 383과 PEP 529). 인자 구문 분석 중에 파일 이름을 bytes로 인코딩하려면, "O&" 변환기를 사용하고 `PyUnicode_FSConverter()`를 변환 함수로 전달해야 합니다:

`int PyUnicode_FSConverter(PyObject* obj, void* result)`

ParseTuple 변환기: (직접 또는 `os.PathLike` 인터페이스를 통해 얻은) `str` 객체를 `PyUnicode_EncodeFSDefault()`를 사용하여 bytes로 인코딩합니다; bytes 객체는 있는 그대로의 출력입니다. *result*는 더는 사용되지 않을 때 해제해야 하는 `PyBytesObject*`여야 합니다.

버전 3.1에 추가.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

인자 구문 분석 중에 파일 이름을 `str`로 디코딩하려면, "O&" 변환기를 사용하고 `PyUnicode_FSDecoder()`를 변환 함수로 전달해야 합니다:

`int PyUnicode_FSDecoder(PyObject* obj, void* result)`

ParseTuple 변환기: (직접 또는 `os.PathLike` 인터페이스를 통해 간접적으로 얻은) bytes 객체를 `PyUnicode_DecodeFSDefaultAndSize()`를 사용하여 `str`로 디코딩합니다; `str` 객체는 있는 그대로의 출력입니다. *result*는 더는 사용되지 않을 때 해제해야 하는 `PyUnicodeObject*`여야 합니다.

버전 3.2에 추가.

버전 3.6에서 변경: 경로류 객체를 받아들입니다.

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

Return value: New reference. `Py_FileSystemDefaultEncoding`과 `Py_FileSystemDefaultEncodeErrors` 에러 처리기를 사용하여 문자열을 디코딩합니다.

`Py_FileSystemDefaultEncoding`이 설정되지 않으면, 로케일 인코딩으로 폴백합니다.

`Py_FileSystemDefaultEncoding`은 시작 시 로케일 인코딩에서 초기화되며 나중에 수정할 수 없습니다. 현재 로케일 인코딩에서 문자열을 디코딩해야 하면, `PyUnicode_DecodeLocaleAndSize()`를 사용하십시오.

더 보기:

`Py_DecodeLocale()` 함수.

버전 3.6에서 변경: `Py_FileSystemDefaultEncodeErrors` 에러 처리기를 사용합니다.

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

Return value: New reference. `Py_FileSystemDefaultEncoding` 과 `Py_FileSystemDefaultEncodeErrors` 에러 처리기를 사용하여 널 종료 문자열을 디코딩합니다.

`Py_FileSystemDefaultEncoding` 이 설정되지 않으면, 로케일 인코딩으로 폴백합니다.

문자열 길이를 알고 있으면 `PyUnicode_DecodeFSDefaultAndSize()` 를 사용하십시오.

버전 3.6에서 변경: `Py_FileSystemDefaultEncodeErrors` 에러 처리기를 사용합니다.

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

Return value: New reference. `Py_FileSystemDefaultEncodeErrors` 에러 처리기를 사용하여 유니코드 객체를 `Py_FileSystemDefaultEncoding` 로 인코딩하고, bytes를 반환합니다. 결과 bytes 객체에는 널 바이트가 포함될 수 있음에 유의하십시오.

`Py_FileSystemDefaultEncoding` 이 설정되지 않으면, 로케일 인코딩으로 폴백합니다.

`Py_FileSystemDefaultEncoding` 은 시작 시 로케일 인코딩에서 초기화되며 나중에 수정할 수 없습니다. 현재 로케일 인코딩으로 문자열을 인코딩해야 하면, `PyUnicode_EncodeLocale()` 을 사용하십시오.

더 보기:

`Py_EncodeLocale()` 함수.

버전 3.2에 추가.

버전 3.6에서 변경: `Py_FileSystemDefaultEncodeErrors` 에러 처리기를 사용합니다.

wchar_t 지원

지원하는 플랫폼에 대한 `wchar_t` 지원:

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

Return value: New reference. 주어진 `size` 의 `wchar_t` 버퍼 `w` 에서 유니코드 객체를 만듭니다. -1을 `size` 로 전달하면 함수가 `wcslen` 을 사용하여 길이를 스스로 계산해야 함을 나타냅니다. 실패하면 NULL을 반환합니다.

`Py_ssize_t` **PyUnicode_AsWideChar** (*PyObject* *unicode, wchar_t *w, `Py_ssize_t` size)

유니코드 객체 내용을 `wchar_t` 버퍼 `w` 에 복사합니다. 최대 `size` `wchar_t` 문자가 복사됩니다 (후행 널 종료 문자가 제외될 수 있습니다). 복사된 `wchar_t` 문자 수나 에러가 발생하면 -1을 반환합니다. 결과 `wchar_t*` 문자열은 널로 종료될 수도 있고 아닐 수도 있음에 유의하십시오. 응용 프로그램에 필요하다면 `wchar_t*` 문자열이 널로 끝나는지 확인하는 것은 호출자의 책임입니다. 또한, `wchar_t*` 문자열에는 널 문자가 포함될 수 있으며, 이로 인해 대부분의 C 함수와 함께 사용될 때 문자열이 잘리게 됨에 유의하십시오.

`wchar_t*` **PyUnicode_AsWideCharString** (*PyObject* *unicode, `Py_ssize_t` *size)

유니코드 객체를 와이드 문자 문자열로 변환합니다. 출력 문자열은 항상 널 문자로 끝납니다. `size` 가 NULL 이 아니면, (후행 널 종료 문자를 제외한) 와이드 문자의 수를 `*size` 에 씁니다. 결과 `wchar_t` 문자열이 널 문자를 포함할 수 있고, 이로 인해 대부분의 C 함수와 함께 사용될 때 문자열이 잘리게 됨에 유의하십시오. `size` 가 NULL 이고 `wchar_t*` 문자열이 널 문자를 포함하면 `ValueError` 가 발생합니다.

성공 시 `PyMem_Alloc()` 에 의해 할당된 버퍼를 반환합니다 (`PyMem_Free()` 를 사용하여 해제하십시오). 에러 시, NULL을 반환하고 `*size` 는 정의되지 않습니다. 메모리 할당이 실패하면 `MemoryError` 를 발생시킵니다.

버전 3.2에 추가.

버전 3.7에서 변경: `size` 가 NULL 이고 `wchar_t*` 문자열이 널 문자를 포함하면 `ValueError` 를 발생시킵니다.

내장 코덱

파이썬은 속도를 위해 C로 작성된 내장 코덱 집합을 제공합니다. 이러한 코덱들은 모두 다음 함수들을 통해 직접 사용할 수 있습니다.

다음 API의 대부분은 두 개의 인자 encoding과 errors를 취하며, 내장 `str()` 문자열 객체 생성자의 것들과 같은 의미입니다.

Setting encoding to NULL causes the default encoding to be used which is ASCII. 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`).

에러 처리는 errors로 설정되는데, 코덱에 대해 정의된 기본 처리를 사용함을 의미하는 NULL로 설정될 수도 있습니다. 모든 내장 코덱에 대한 기본 에러 처리는 “strict” 입니다 (ValueError가 발생합니다).

코덱은 모두 유사한 인터페이스를 사용합니다. 단순성을 위해 다음에 나오는 일반 코덱과의 차이만 설명합니다.

일반 코덱

다음은 일반 코덱 API입니다:

PyObject* PyUnicode_Decode (const char *s, Py_ssize_t size, const char *encoding, const char *errors)
Return value: New reference. 인코딩된 문자열 s의 size 바이트를 디코딩하여 유니코드 객체를 만듭니다. encoding과 errors는 `str()` 내장 함수의 같은 이름의 매개 변수와 같은 의미입니다. 사용할 코덱은 파이썬 코덱 레지스트리를 사용하여 조회됩니다. 코덱에서 예외가 발생하면 NULL을 반환합니다.

PyObject* PyUnicode_AsEncodedString (PyObject *unicode, const char *encoding, const char *errors)
Return value: New reference. 유니코드 객체를 인코딩하고 결과를 파이썬 bytes 객체로 반환합니다. encoding과 errors는 유니코드 `encode()` 메서드의 같은 이름의 매개 변수와 같은 의미입니다. 사용할 코덱은 파이썬 코덱 레지스트리를 사용하여 조회됩니다. 코덱에서 예외가 발생하면 NULL을 반환합니다.

PyObject* PyUnicode_Encode (const Py_UNICODE *s, Py_ssize_t size, const char *encoding, const char *errors)
Return value: New reference. 주어진 size의 Py_UNICODE 버퍼 s를 인코딩하고 파이썬 bytes 객체를 반환합니다. encoding과 errors는 유니코드 `encode()` 메서드의 같은 이름의 매개 변수와 같은 의미입니다. 사용할 코덱은 파이썬 코덱 레지스트리를 사용하여 조회됩니다. 코덱에서 예외가 발생하면 NULL을 반환합니다.

Deprecated since version 3.3, will be removed in version 3.11: 이전 스타일 `Py_UNICODE` API의 일부입니다; `PyUnicode_AsEncodedString()` 을 사용하여 마이그레이션 하십시오.

UTF-8 코덱

다음은 UTF-8 코덱 API입니다:

PyObject* PyUnicode_DecodeUTF8 (const char *s, Py_ssize_t size, const char *errors)
Return value: New reference. UTF-8로 인코딩된 문자열 s의 size 바이트를 디코딩하여 유니코드 객체를 만듭니다. 코덱에서 예외가 발생하면 NULL을 반환합니다.

PyObject* PyUnicode_DecodeUTF8Stateful (const char *s, Py_ssize_t size, const char *errors, Py_ssize_t *consumed)
Return value: New reference. consumed가 NULL이면, `PyUnicode_DecodeUTF8()` 처럼 동작합니다. consumed가 NULL이 아니면, 후행 불완전한 UTF-8 바이트 시퀀스는 예외로 처리되지 않습니다. 이러한 바이트는 디코딩되지 않으며 디코딩된 바이트 수는 consumed에 저장됩니다.

*PyObject** **PyUnicode_AsUTF8String** (*PyObject* *unicode)

Return value: New reference. UTF-8을 사용하여 유니코드 객체를 인코딩하고 결과를 파이썬 bytes 객체로 반환합니다. 에러 처리는 “strict” 입니다. 코덱에서 예외가 발생하면 NULL을 반환합니다.

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

유니코드 객체의 UTF-8 인코딩에 대한 포인터를 반환하고, 인코딩된 표현의 크기를 (바이트 단위로) *size*에 저장합니다. *size* 인자는 NULL일 수 있습니다; 이 경우 크기가 저장되지 않습니다. 반환된 버퍼에는 다른 널 코드 포인트가 있는지에 관계없이, 항상 추가 널 바이트가 추가됩니다 (*size*에 포함되지 않습니다).

에러가 발생하면, NULL이 예외 설정과 함께 반환되고 *size*가 저장되지 않습니다.

이것은 유니코드 객체에서 문자열의 UTF-8 표현을 캐시하고, 후속 호출은 같은 버퍼에 대한 포인터를 반환합니다. 호출자는 버퍼 할당 해제에 대한 책임이 없습니다.

버전 3.3에 추가.

버전 3.7에서 변경: 반환형은 이제 char *가 아니라 const char *입니다.

const char* **PyUnicode_AsUTF8** (*PyObject* *unicode)

PyUnicode_AsUTF8AndSize()와 같지만, 크기를 저장하지 않습니다.

버전 3.3에 추가.

버전 3.7에서 변경: 반환형은 이제 char *가 아니라 const char *입니다.

*PyObject** **PyUnicode_EncodeUTF8** (const *Py_UNICODE* *s, Py_ssize_t size, const char *errors)

Return value: New reference. UTF-8을 사용하여 주어진 *size*의 *Py_UNICODE* 버퍼 *s*를 인코딩하고 파이썬 bytes 객체를 반환합니다. 코덱에서 예외가 발생하면 NULL을 반환합니다.

Deprecated since version 3.3, will be removed in version 3.11: 이전 스타일 *Py_UNICODE* API의 일부입니다; *PyUnicode_AsUTF8String*(), *PyUnicode_AsUTF8AndSize*() 나 *PyUnicode_AsEncodedString*()을 사용하여 마이그레이션 하십시오.

UTF-32 코덱

다음은 UTF-32 코덱 API입니다:

*PyObject** **PyUnicode_DecodeUTF32** (const char *s, Py_ssize_t size, const char *errors, int *byteorder)

Return value: New reference. UTF-32로 인코딩된 버퍼 문자열에서 *size* 바이트를 디코딩하고 해당 유니코드 객체를 반환합니다. *errors*(NULL이 아니면)는 에러 처리를 정의합니다. 기본값은 “strict”입니다.

*byteorder*가 NULL이 아니면, 디코더는 지정된 바이트 순서를 사용하여 디코딩을 시작합니다:

```
*byteorder == -1: little endian
*byteorder == 0:  native order
*byteorder == 1:  big endian
```

*byteorder가 0이고, 입력 데이터의 처음 4바이트가 바이트 순서 표시(BOM)이면, 디코더가 이 바이트 순서로 전환되고 BOM은 결과 유니코드 문자열에 복사되지 않습니다. *byteorder가 -1이나 1이면, 모든 바이트 순서 표시가 출력에 복사됩니다.

완료 후, *byteorder는 입력 데이터의 끝에서 현재 바이트 순서로 설정됩니다.

*byteorder*가 NULL이면, 코덱은 네이티브 순서 모드로 시작합니다.

코덱에서 예외가 발생하면 NULL을 반환합니다.

*PyObject** **PyUnicode_DecodeUTF32Stateful** (const char *s, Py_ssize_t size, const char *errors, int *byteorder, Py_ssize_t *consumed)

Return value: New reference. *consumed*가 NULL이면, *PyUnicode_DecodeUTF32*()처럼 동작합니다. *consumed*가 NULL이 아니면, *PyUnicode_DecodeUTF32Stateful*()은 후행 불완전 UTF-32 바이트 시퀀스

스(가령 4로 나누어떨어지지 않는 바이트 수)를 예외로 처리하지 않습니다. 이러한 바이트는 디코딩되지 않으며 디코딩된 바이트 수는 *consumed*에 저장됩니다.

PyObject* PyUnicode_AsUTF32String (PyObject *unicode)

Return value: New reference. 네이티브 바이트 순서로 UTF-32 인코딩을 사용하여 파이썬 바이트 문자열을 반환합니다. 문자열은 항상 BOM 마크로 시작합니다. 예외 처리는 “strict”입니다. 코덱에서 예외가 발생하면 NULL을 반환합니다.

PyObject* PyUnicode_EncodeUTF32 (const Py_UNICODE *s, Py_ssize_t size, const char *errors, int byteorder)

Return value: New reference. *s*에 있는 유니코드 데이터의 UTF-32로 인코딩된 값을 포함하는 파이썬 bytes 객체를 반환합니다. 출력은 다음 바이트 순서에 따라 기록됩니다:

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

byteorder가 0이면, 출력 문자열은 항상 유니코드 BOM 마크(U+FEFF)로 시작합니다. 다른 두 모드에서는, BOM 마크가 앞에 추가되지 않습니다.

Py_UNICODE_WIDE가 정의되지 않으면, 서로게이트 쌍이 단일 코드 포인트로 출력됩니다.

코덱에서 예외가 발생하면 NULL을 반환합니다.

Deprecated since version 3.3, will be removed in version 3.11: 이전 스타일 *Py_UNICODE* API의 일부입니다. *PyUnicode_AsUTF32String()* 이나 *PyUnicode_AsEncodedString()* 을 사용하여 마이그레이션 하십시오.

UTF-16 코덱

다음은 UTF-16 코덱 API입니다:

PyObject* PyUnicode_DecodeUTF16 (const char *s, Py_ssize_t size, const char *errors, int *byteorder)

Return value: New reference. UTF-16으로 인코딩된 버퍼 문자열에서 *size* 바이트를 디코딩하고 해당 유니코드 객체를 반환합니다. *errors*(NULL이 아니면)는 예외 처리를 정의합니다. 기본값은 “strict”입니다.

*byteorder*가 NULL이 아니면, 디코더는 지정된 바이트 순서를 사용하여 디코딩을 시작합니다:

```
*byteorder == -1: little endian
*byteorder == 0:  native order
*byteorder == 1:  big endian
```

*byteorder가 0이고, 입력 데이터의 처음 2바이트가 바이트 순서 표시(BOM)이면, 디코더는 이 바이트 순서로 전환되고 BOM은 결과 유니코드 문자열에 복사되지 않습니다. *byteorder가 -1이나 1이면 모든 바이트 순서 표시가 출력에 복사됩니다(\ufeff나 \ufffe 문자가 됩니다).

완료 후, *byteorder는 입력 데이터의 끝에서 현재 바이트 순서로 설정됩니다.

*byteorder*가 NULL이면, 코덱은 네이티브 순서 모드로 시작합니다.

코덱에서 예외가 발생하면 NULL을 반환합니다.

PyObject* PyUnicode_DecodeUTF16Stateful (const char *s, Py_ssize_t size, const char *errors, int *byteorder, Py_ssize_t *consumed)

Return value: New reference. *consumed*가 NULL이면, *PyUnicode_DecodeUTF16()* 처럼 동작합니다. *consumed*가 NULL이 아니면, *PyUnicode_DecodeUTF16Stateful()* 은 후행 불완전 UTF-16 바이트 시퀀스(가령 홀수 바이트 수나 분할 서로게이트 쌍)를 예외로 처리하지 않습니다. 이러한 바이트는 디코딩되지 않으며 디코딩된 바이트 수는 *consumed*에 저장됩니다.

*PyObject** **PyUnicode_AsUTF16String** (*PyObject* *unicode)

Return value: New reference. 네이티브 바이트 순서로 UTF-16 인코딩을 사용하여 파이썬 바이트 문자열을 반환합니다. 문자열은 항상 BOM 마크로 시작합니다. 에러 처리는 “strict”입니다. 코덱에서 예외가 발생하면 NULL을 반환합니다.

*PyObject** **PyUnicode_EncodeUTF16** (const *Py_UNICODE* *s, Py_ssize_t size, const char *errors, int byte-order)

Return value: New reference. s에 있는 유니코드 데이터의 UTF-16 인코딩된 값을 포함하는 파이썬 bytes 객체를 반환합니다. 출력은 다음 바이트 순서에 따라 기록됩니다:

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

byteorder가 0이면, 출력 문자열은 항상 유니코드 BOM 마크(U+FEFF)로 시작합니다. 다른 두 모드에서는, BOM 마크가 앞에 추가되지 않습니다.

Py_UNICODE_WIDE가 정의되면, 단일 *Py_UNICODE* 값이 서로게이트 쌍으로 표시될 수 있습니다. 정의되지 않으면, 각 *Py_UNICODE* 값은 UCS-2 문자로 해석됩니다.

코덱에서 예외가 발생하면 NULL을 반환합니다.

Deprecated since version 3.3, will be removed in version 3.11: 이전 스타일 *Py_UNICODE* API의 일부입니다; *PyUnicode_AsUTF16String()* 이나 *PyUnicode_AsEncodedString()* 을 사용하여 마이그레이션 하십시오.

UTF-7 코덱

다음은 UTF-7 코덱 API입니다:

*PyObject** **PyUnicode_DecompileUTF7** (const char *s, Py_ssize_t size, const char *errors)

Return value: New reference. UTF-7로 인코딩된 문자열 s의 size 바이트를 디코딩하여 유니코드 객체를 만듭니다. 코덱에서 예외가 발생하면 NULL을 반환합니다.

*PyObject** **PyUnicode_DecompileUTF7Stateful** (const char *s, Py_ssize_t size, const char *errors, Py_ssize_t *consumed)

Return value: New reference. consumed가 NULL이면, *PyUnicode_DecompileUTF7()* 처럼 동작합니다. consumed가 NULL이 아니면, 후행 불완전한 UTF-7 base-64 섹션은 에러로 처리되지 않습니다. 이러한 바이트는 디코딩되지 않으며 디코딩된 바이트 수는 consumed에 저장됩니다.

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

Return value: New reference. UTF-7을 사용하여 주어진 크기의 *Py_UNICODE* 버퍼를 인코딩하고 파이썬 bytes 객체를 반환합니다. 코덱에서 예외가 발생하면 NULL을 반환합니다.

base64SetO 가 0이 아니면, “Set O”(다른 특별한 의미가 없는 구두점)는 base-64로 인코딩됩니다. base64WhiteSpace 가 0이 아니면, 공백은 base-64로 인코딩됩니다. 파이썬 “utf-7” 코덱의 경우 둘 다 0으로 설정됩니다.

Deprecated since version 3.3, will be removed in version 3.11: 이전 스타일 *Py_UNICODE* API의 일부입니다; *PyUnicode_AsEncodedString()* 을 사용하여 마이그레이션 하십시오.

유니코드 이스케이프 코덱

다음은 “유니코드 이스케이프(Unicode Escape)” 코덱 API입니다:

PyObject* PyUnicode_DecodeUnicodeEscape (const char *s, Py_ssize_t size, const char *errors)
Return value: New reference. 유니코드 이스케이프 인코딩된 문자열 s의 size 바이트를 디코딩하여 유니코드 객체를 만듭니다. 코덱에서 예외가 발생하면 NULL을 반환합니다.

PyObject* PyUnicode_AsUnicodeEscapeString (PyObject *unicode)
Return value: New reference. 유니코드 이스케이프를 사용하여 유니코드 객체를 인코딩하고 결과를 bytes 객체로 반환합니다. 에러 처리는 “strict”입니다. 코덱에서 예외가 발생하면 NULL을 반환합니다.

PyObject* PyUnicode_EncodeUnicodeEscape (const Py_UNICODE *s, Py_ssize_t size)
Return value: New reference. 유니코드 이스케이프를 사용하여 주어진 size의 Py_UNICODE 버퍼를 인코딩하고 bytes 객체를 반환합니다. 코덱에서 예외가 발생하면 NULL을 반환합니다.

Deprecated since version 3.3, will be removed in version 3.11: 이전 스타일 Py_UNICODE API의 일부입니다; PyUnicode_AsUnicodeEscapeString() 을 사용하여 마이그레이션 하십시오.

원시 유니코드 이스케이프 코덱

다음은 “원시 유니코드 이스케이프(Raw Unicode Escape)” 코덱 API입니다:

PyObject* PyUnicode_DecodeRawUnicodeEscape (const char *s, Py_ssize_t size, const char *errors)
Return value: New reference. 원시 유니코드 이스케이프 인코딩된 문자열 s의 size 바이트를 디코딩하여 유니코드 객체를 만듭니다. 코덱에서 예외가 발생하면 NULL을 반환합니다.

PyObject* PyUnicode_AsRawUnicodeEscapeString (PyObject *unicode)
Return value: New reference. 원시 유니코드 이스케이프를 사용하여 유니코드 객체를 인코딩하고 결과를 bytes 객체로 반환합니다. 에러 처리는 “strict”입니다. 코덱에서 예외가 발생하면 NULL을 반환합니다.

PyObject* PyUnicode_EncodeRawUnicodeEscape (const Py_UNICODE *s, Py_ssize_t size)
Return value: New reference. 원시 유니코드 이스케이프를 사용하여 주어진 size의 Py_UNICODE 버퍼를 인코딩하고 bytes 객체를 반환합니다. 코덱에서 예외가 발생하면 NULL을 반환합니다.

Deprecated since version 3.3, will be removed in version 3.11: 이전 스타일 Py_UNICODE API의 일부입니다; PyUnicode_AsRawUnicodeEscapeString() 이나 PyUnicode_AsEncodedString() 을 사용하여 마이그레이션 하십시오.

Latin-1 코덱

다음은 Latin-1 코덱 API입니다: Latin-1은 처음 256개의 유니코드 서수에 해당하며 인코딩 중에 코덱에서 이들만 허용됩니다.

PyObject* PyUnicode_DecodeLatin1 (const char *s, Py_ssize_t size, const char *errors)
Return value: New reference. Latin-1 인코딩된 문자열 s의 size 바이트를 디코딩하여 유니코드 객체를 만듭니다. 코덱에서 예외가 발생하면 NULL을 반환합니다.

PyObject* PyUnicode_AsLatin1String (PyObject *unicode)
Return value: New reference. Latin-1을 사용하여 유니코드 객체를 인코딩하고 결과를 파이썬 bytes 객체로 반환합니다. 에러 처리는 “strict”입니다. 코덱에서 예외가 발생하면 NULL을 반환합니다.

PyObject* PyUnicode_EncodeLatin1 (const Py_UNICODE *s, Py_ssize_t size, const char *errors)
Return value: New reference. Latin-1을 사용하여 주어진 size의 Py_UNICODE 버퍼를 인코딩하고 파이썬 bytes 객체를 반환합니다. 코덱에서 예외가 발생하면 NULL을 반환합니다.

Deprecated since version 3.3, will be removed in version 3.11: 이전 스타일 `Py_UNICODE` API의 일부입니다; `PyUnicode_AsLatin1String()` 이나 `PyUnicode_AsEncodedString()` 을 사용하여 마이그레이션 하십시오.

ASCII 코덱

다음은 ASCII 코덱 API입니다. 7비트 ASCII 데이터만 허용됩니다. 다른 모든 코드는 에러를 생성합니다.

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

Return value: New reference. ASCII 인코딩된 문자열 s의 size 바이트를 디코딩하여 유니코드 객체를 만듭니다. 코덱에서 예외가 발생하면 NULL을 반환합니다.

PyObject* PyUnicode_AsASCIIString (PyObject *unicode)

Return value: New reference. ASCII를 사용하여 유니코드 객체를 인코딩하고 결과를 파이썬 bytes 객체로 반환합니다. 에러 처리는 “strict”입니다. 코덱에서 예외가 발생하면 NULL을 반환합니다.

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

Return value: New reference. ASCII를 사용하여 주어진 size의 `Py_UNICODE` 버퍼를 인코딩하고 파이썬 bytes 객체를 반환합니다. 코덱에서 예외가 발생하면 NULL을 반환합니다.

Deprecated since version 3.3, will be removed in version 3.11: 이전 스타일 `Py_UNICODE` API의 일부입니다; `PyUnicode_AsASCIIString()` 이나 `PyUnicode_AsEncodedString()` 을 사용하여 마이그레이션 하십시오.

문자 맵 코덱

이 코덱은 다양한 코덱을 구현하는 데 사용할 수 있다는 점에서 특별합니다 (실제로 encodings 패키지에 포함된 대부분의 표준 코덱을 얻기 위해 수행되었습니다). 코덱은 매핑을 사용하여 문자를 인코딩하고 디코딩합니다. 제공된 매핑 객체는 `__getitem__()` 매핑 인터페이스를 지원해야 합니다; 딕셔너리와 시퀀스가 잘 작동합니다.

다음은 매핑 코덱 API입니다:

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

Return value: New reference. 주어진 mapping 객체를 사용하여 인코딩된 문자열 s의 size 바이트를 디코딩하여 유니코드 객체를 만듭니다. 코덱에서 예외가 발생하면 NULL을 반환합니다.

mapping이 NULL이면, Latin-1 디코딩이 적용됩니다. 그렇지 않으면 mapping은 바이트 서수(0에서 255 사이의 정수)를 유니코드 문자열, 정수(유니코드 서수로 해석됩니다) 또는 None으로 매핑해야 합니다. 매핑되지 않은 데이터 바이트(None, 0xFFFE 또는 '\ufffe'로 매핑되는 것뿐만 아니라, LookupError를 유발하는 것)은 정의되지 않은 매핑으로 처리되어 에러를 발생시킵니다.

PyObject* PyUnicode_AsCharmapString (PyObject *unicode, PyObject *mapping)

Return value: New reference. 주어진 mapping 객체를 사용하여 유니코드 객체를 인코딩하고 결과를 bytes 객체로 반환합니다. 에러 처리는 “strict”입니다. 코덱에서 예외가 발생하면 NULL을 반환합니다.

mapping 객체는 유니코드 서수 정수를 bytes 객체, 0에서 255 사이의 정수 또는 None으로 매핑해야 합니다. None에 매핑되는 것뿐만 아니라 매핑되지 않은 문자 서수(LookupError를 유발하는 것)는 “정의되지 않은 매핑”으로 처리되어 에러가 발생합니다.

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

Return value: New reference. 주어진 mapping 객체를 사용하여 주어진 size의 `Py_UNICODE` 버퍼를 인코딩하고 그 결과를 bytes 객체로 반환합니다. 코덱에서 예외가 발생하면 NULL을 반환합니다.

Deprecated since version 3.3, will be removed in version 3.11: 이전 스타일 `Py_UNICODE` API의 일부입니다; `PyUnicode_AsCharmapString()` 이나 `PyUnicode_AsEncodedString()` 을 사용하여 마이그레이션 하십시오.

다음 코덱 API는 유니코드를 유니코드로 매핑한다는 점에서 특별합니다.

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

Return value: New reference. 문자 매핑 테이블을 적용하여 문자열을 변환하고 결과 유니코드 객체를 반환합니다. 코덱에서 예외가 발생하면 NULL을 반환합니다.

매핑 테이블은 유니코드 서수 정수를 유니코드 서수 정수나 None(문자가 삭제되도록 합니다)에 매핑해야 합니다.

매핑 테이블은 `__getitem__()` 인터페이스 만 제공하면 됩니다; 딕셔너리와 시퀀스가 잘 작동합니다. 매핑되지 않은 문자 서수(LookupError를 유발하는 것)는 건드리지 않고 그대로 복사됩니다.

*errors*는 코덱에서의 일반적인 의미입니다. 기본 에러 처리를 사용함을 나타내는 NULL일 수 있습니다.

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

Return value: New reference. 문자 *mapping* 테이블을 적용하여 주어진 *size*의 `Py_UNICODE` 버퍼를 변환하고 결과 유니코드 객체를 반환합니다. 코덱에서 예외가 발생하면 NULL을 반환합니다.

Deprecated since version 3.3, will be removed in version 3.11: 이전 스타일 `Py_UNICODE` API의 일부입니다; `PyUnicode_Translate()` 나 일반 코덱 기반 API를 사용하여 마이그레이션 하십시오.

윈도우 용 MBCS 코덱

다음은 MBCS 코덱 API입니다. 현재 윈도우에서만 사용할 수 있으며 Win32 MBCS 변환기를 사용하여 변환을 구현합니다. MBCS(또는 DBCS)는 단지 하나가 아니라 인코딩 클래스임에 유의하십시오. 대상 인코딩은 코덱을 실행하는 기계의 사용자 설정에 의해 정의됩니다.

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

Return value: New reference. MBCS 인코딩된 문자열 *s*의 *size* 바이트를 디코딩하여 유니코드 객체를 만듭니다. 코덱에서 예외가 발생하면 NULL을 반환합니다.

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

Return value: New reference. *consumed*가 NULL이면, `PyUnicode_DecodeMBCS()` 처럼 동작합니다. *consumed*가 NULL이 아니면, `PyUnicode_DecodeMBCSStateful()` 은 후행 선행(lead) 바이트를 디코딩하지 않고 디코딩된 바이트 수가 *consumed*에 저장됩니다.

PyObject* PyUnicode_AsMBCSString (PyObject *unicode)

Return value: New reference. MBCS를 사용하여 유니코드 객체를 인코딩하고 결과를 파이썬 bytes 객체로 반환합니다. 에러 처리는 “strict”입니다. 코덱에서 예외가 발생하면 NULL을 반환합니다.

PyObject* PyUnicode_EncodeCodePage (int code_page, PyObject *unicode, const char *errors)

Return value: New reference. 지정된 코드 페이지를 사용하여 유니코드 객체를 인코딩하고 파이썬 bytes 객체를 반환합니다. 코덱에서 예외가 발생하면 NULL을 반환합니다. CP_ACP 코드 페이지를 사용하여 MBCS 인코더를 얻습니다.

버전 3.3에 추가.

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

Return value: New reference. MBCS를 사용하여 주어진 *size*의 `Py_UNICODE` 버퍼를 인코딩하고 파이썬 bytes 객체를 반환합니다. 코덱에서 예외가 발생하면 NULL을 반환합니다.

Deprecated since version 3.3, will be removed in version 4.0: 이전 스타일 `Py_UNICODE` API의 일부입니다; `PyUnicode_AsMBCSString()`, `PyUnicode_EncodeCodePage()` 또는 `PyUnicode_AsEncodedString()` 을 사용하여 마이그레이션 하십시오.

메서드와 슬롯

메서드와 슬롯 함수

다음 API는 입력의 유니코드 객체와 문자열을 (설명에서 문자열이라고 하겠습니다) 처리할 수 있으며 적절하게 유니코드 객체나 정수를 반환합니다.

예외가 발생하면 모두 NULL이나 -1을 반환합니다.

PyObject* PyUnicode_Concat (PyObject *left, PyObject *right)

Return value: New reference. 두 문자열을 이어붙여 하나의 새로운 유니코드 문자열을 제공합니다.

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

Return value: New reference. 문자열을 분할하여 유니코드 문자열 리스트를 제공합니다. *sep*이 NULL이면, 모든 공백 부분 문자열에서 분할이 수행됩니다. 그렇지 않으면, 주어진 구분자에서 분할이 일어납니다. 최대 *maxsplit* 분할이 수행됩니다. 음수이면, 제한이 설정되지 않습니다. 구분자는 결과 리스트에 포함되지 않습니다.

PyObject* PyUnicode_Splitlines (PyObject *s, int keepend)

Return value: New reference. 줄 바꿈에서 유니코드 문자열을 분할하여, 유니코드 문자열 리스트를 반환합니다. CRLF는 하나의 줄 바꿈으로 간주합니다. *keepend*가 0이면, 결과 문자열에 줄 바꿈 문자가 포함되지 않습니다.

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

Return value: New reference. 주어진 *separator*를 사용하여 문자열 시퀀스를 연결하고 결과 유니코드 문자열을 반환합니다.

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

*substr*이 주어진 꼬리 끝에서 (*direction* == -1은 접두사 일치를 수행함을 의미하고, *direction* == 1은 접미사 일치를 의미합니다) *str*[start:end]와 일치하면 1을 반환합니다, 그렇지 않으면 0을 반환합니다. 에러가 발생하면 -1을 반환합니다.

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

주어진 *direction*을 사용하여 (*direction* == 1은 정방향 검색을 의미하고, *direction* == -1은 역방향 검색을 의미합니다) *str*[start:end]에서 *substr*의 첫 번째 위치를 반환합니다. 반환 값은 첫 번째 일치의 인덱스입니다; -1 값은 일치하는 항목이 없음을 나타내고, -2는 에러가 발생했고 예외가 설정되었음을 나타냅니다.

Py_ssize_t PyUnicode_FindChar (PyObject *str, Py_UCS4 ch, Py_ssize_t start, Py_ssize_t end, int direction)

주어진 *direction*을 사용하여 (*direction* == 1은 정방향 검색을 의미하고, *direction* == -1은 역방향 검색을 의미합니다) *str*[start:end]에서 문자 *ch*의 첫 번째 위치를 반환합니다. 반환 값은 첫 번째 일치의 인덱스입니다; -1 값은 일치하는 항목이 없음을 나타내고, -2는 에러가 발생했고 예외가 설정되었음을 나타냅니다.

버전 3.3에 추가.

버전 3.7에서 변경: *start*와 *end*는 이제 *str*[start:end]처럼 작동하도록 조정됩니다.

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

str[start:end]에서 *substr*이 겹치지 않게 등장하는 횟수를 반환합니다. 에러가 발생하면 -1을 반환합니다.

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

Return value: New reference. *str*에서 *substr*의 최대 *maxcount* 등장을 *replstr*로 바꾸고 결과 유니코드 객체를 반환합니다. *maxcount* == -1은 모든 등장을 교체함을 의미합니다.

int PyUnicode_Compare (PyObject *left, PyObject *right)

두 문자열을 비교하고 각각 작음, 같음, 큼에 대해 -1, 0, 1을 반환합니다.

이 함수는 실패 시 -1을 반환하므로, 에러를 확인하기 위해 *PyErr_Occurred()*를 호출해야 합니다.

int **PyUnicode_CompareWithASCIIString** (*PyObject* *uni, const char *string)

유니코드 객체 *uni*를 *string*과 비교하고 각각 작음, 같음, 큼에 대해 -1, 0, 1을 반환합니다. ASCII로 인코딩된 문자열만 전달하는 것이 가장 좋지만, 비 ASCII 문자가 포함되면 함수는 입력 문자열을 ISO-8859-1로 해석합니다.

이 함수는 예외를 발생시키지 않습니다.

*PyObject** **PyUnicode_RichCompare** (*PyObject* *left, *PyObject* *right, int op)

Return value: New reference. 두 유니코드 문자열을 풍부한 비교 (rich comparison) 하고 다음 중 하나를 반환합니다:

- 예외가 발생하면 NULL
- 성공적인 비교는 Py_True나 Py_False
- 형 조합을 알 수 없으면 Py_NotImplemented

op에 가능한 값은 Py_GT, Py_GE, Py_EQ, Py_NE, Py_LT 및 Py_LE입니다.

*PyObject** **PyUnicode_Format** (*PyObject* *format, *PyObject* *args)

Return value: New reference. *format*과 *args*에서 새 문자열 객체를 반환합니다; 이것은 *format % args*와 유사합니다.

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

*element*가 *container*에 포함되어 있는지 확인하고 그에 따라 참이나 거짓을 반환합니다.

*element*는 단일 요소 유니코드 문자열로 강제 변환해야 합니다. 예러가 있으면 -1이 반환됩니다.

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

인자 **string*을 제자리에서 인턴(intern) 합니다. 인자는 파이썬 유니코드 문자열을 가리키는 포인터 변수의 주소여야 합니다. **string*과 같은 기존 인턴 문자열이 있으면, **string*을 그것으로 설정합니다 (이전 문자열 객체의 참조 횟수를 감소시키고 인턴된 문자열 객체의 참조 횟수를 증가시킵니다), 그렇지 않으면 **string*만 홀로 두고 인턴 합니다 (참조 횟수를 증가시킵니다). (설명: 참조 횟수에 대해 많은 이야기가 있지만, 이 함수를 참조 횟수 중첩이라고 생각하십시오; 호출 전에 소유한 경우에만 호출 후 객체를 소유합니다.)

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

Return value: New reference. *PyUnicode_FromString()*과 *PyUnicode_InternInPlace()*의 조합, 인턴(intern)된 새 유니코드 문자열 객체나, 같은 값을 가진 이전에 인턴된 문자열 객체에 대한 새 ("소유된") 참조를 반환합니다.

8.3.4 튜플 객체

PyTupleObject

이 *PyObject*의 서브 형은 파이썬 튜플 객체를 나타냅니다.

PyTypeObject **PyTuple_Type**

이 *PyTypeObject* 인스턴스는 파이썬 튜플 형을 나타냅니다. 파이썬 계층의 tuple과 같은 객체입니다.

int **PyTuple_Check** (*PyObject* *p)

*p*가 튜플 객체이거나 튜플 형의 서브 형의 인스턴스면 참을 돌려줍니다.

int **PyTuple_CheckExact** (*PyObject* *p)

*p*가 튜플 객체이지만, 튜플 형의 서브 형의 인스턴스는 아니면 참을 돌려줍니다.

*PyObject** **PyTuple_New** (Py_ssize_t len)

Return value: New reference. 크기 *len* 인 튜플 객체나, 실패 시 NULL을 반환합니다.

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

Return value: New reference. 크기 *n* 인 새 튜플 객체나, 실패 시 NULL을 반환합니다. 튜플 값은 파이썬 객체를 가리키는 후속 *n* 개의 C 인자로 초기화됩니다. *PyTuple_Pack*(2, *a*, *b*)는 *Py_BuildValue*("(OO)", *a*, *b*)와 동등합니다.

`Py_ssize_t PyTuple_Size (PyObject *p)`

튜플 객체에 대한 포인터를 받아서, 해당 튜플의 크기를 반환합니다.

`Py_ssize_t PyTuple_GET_SIZE (PyObject *p)`

튜플 *p*의 크기를 반환합니다. 이 크기는 NULL이 아니고 튜플을 가리켜야 합니다; 에러 검사는 수행되지 않습니다.

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

Return value: Borrowed reference. *p*가 가리키는 튜플의 *pos* 위치에 있는 객체를 반환합니다. *pos*가 범위를 벗어나면, NULL을 반환하고 `IndexError` 예외를 설정합니다.

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

Return value: Borrowed reference. `PyTuple_GetItem()`와 비슷하지만, 인자를 확인하지 않습니다.

`PyObject* PyTuple_GetSlice (PyObject *p, Py_ssize_t low, Py_ssize_t high)`

Return value: New reference. *p*가 가리키는 튜플의 *low*와 *high* 사이의 슬라이스를 반환하거나, 실패하면 NULL을 반환합니다. 이것은 파이썬 표현식 `p[low:high]`와 동등합니다. 리스트 끝으로부터의 인덱싱은 지원되지 않습니다.

`int PyTuple_SetItem (PyObject *p, Py_ssize_t pos, PyObject *o)`

*p*가 가리키는 튜플의 *pos* 위치에 객체 *o*에 대한 참조를 삽입합니다. 성공하면 0을 반환합니다. *pos*가 범위를 벗어나면, -1을 반환하고 `IndexError` 예외를 설정합니다.

참고: 이 함수는 *o*에 대한 참조를 “훔치고” 영향을 받는 위치에서 튜플에 이미 있는 항목에 대한 참조를 버립니다(discard).

`void PyTuple_SET_ITEM (PyObject *p, Py_ssize_t pos, PyObject *o)`

`PyTuple_SetItem()`과 비슷하지만, 에러 검사는 하지 않으며 새로운 튜플을 채울 때*만* 사용해야 합니다.

참고: 이 매크로는 *o*에 대한 참조를 “훔치고”, `PyTuple_SetItem()`와 달리, 교체 중인 항목에 대한 참조를 버리지 않습니다; *pos* 위치에서 튜플의 모든 참조는 누수됩니다.

`int _PyTuple_Resize (PyObject **p, Py_ssize_t newsize)`

튜플 크기를 조정하는 데 사용할 수 있습니다. *newsize*는 튜플의 새로운 길이가 됩니다. 튜플은 불변이라고 여겨지므로, 객체에 대해 참조가 하나만 있을 때만 사용해야 합니다. 튜플이 코드의 다른 부분에 이미 알려졌으면 이것을 사용하지 마십시오. 튜플은 항상 끝에서 커지거나 줄어듭니다. 이것을 오래된 튜플을 파괴하고 새 튜플을 만드는 것으로 생각하십시오, 단지 더 효율적일 뿐입니다. 성공하면 0을 반환합니다. 클라이언트 코드는, **p*의 결괏값이 이 함수를 호출하기 전과 같다고 가정해서는 안 됩니다. **p*가 참조하는 객체가 바뀌면 원래 **p*는 파괴됩니다. 실패하면, -1을 반환하고, **p*를 NULL로 설정하고, `MemoryError` 나 `SystemError`를 발생시킵니다.

`int PyTuple_ClearFreeList ()`

자유 목록(free list)을 지웁니다. 해제된 총 항목 수를 반환합니다.

8.3.5 구조체 시퀀스 객체

구조체 시퀀스(struct sequence) 객체는 `namedtuple()` 객체의 C 등가물입니다, 즉 어트리뷰트를 통해 항목에 액세스할 수 있는 시퀀스입니다. 구조체 시퀀스를 만들려면, 먼저 특정 구조체 시퀀스 형을 만들어야 합니다.

*PyObject** **PyStructSequence_NewType** (*PyStructSequence_Desc* *desc)

Return value: New reference. 아래에 설명된 *desc*의 데이터로 새로운 구조체 시퀀스 형을 만듭니다. 결과 형의 인스턴스는 `PyStructSequence_New()`로 만들 수 있습니다.

void **PyStructSequence_InitType** (*PyObject* *type, *PyStructSequence_Desc* *desc)

*desc*로 구조체 시퀀스 형 *type*을 재자리에서 초기화합니다.

int **PyStructSequence_InitType2** (*PyObject* *type, *PyStructSequence_Desc* *desc)

`PyStructSequence_InitType`와 같지만, 성공하면 0을, 실패하면 -1을 반환합니다.

버전 3.4에 추가.

PyStructSequence_Desc

만들 구조체 시퀀스 형의 메타 정보를 포함합니다.

필드	C 형	의미
name	const char *	구조체 시퀀스 형의 이름
doc	const char *	해당 형에 대한 독스트링에 대한 포인터나 생략하려면 NULL
fields	PyStructSequence_Field *	새로운 형의 필드 이름을 가진 NULL로 끝나는 배열에 대한 포인터
n_in_sequence	int	파이썬 측에서 볼 수 있는 필드 수 (튜플로 사용된 경우)

PyStructSequence_Field

구조체 시퀀스의 필드를 기술합니다. 구조체 시퀀스는 튜플로 모형화되므로, 모든 필드는 *PyObject** 형을 취합니다. *PyStructSequence_Desc*의 *fields* 배열의 인덱스는 구조체 시퀀스의 어떤 필드가 기술되는지를 결정합니다.

필드	C 형	의미
name	const char *	필드의 이름이나 이름있는 필드의 목록을 끝내려면 NULL, 이름이 없는 상태로 두려면 <code>PyStructSequence_UnnamedField</code> 로 설정합니다
doc	const char *	필드 독스트링이나 생략하려면 NULL

char* **PyStructSequence_UnnamedField**

이름 없는 상태로 남겨두기 위한 필드 이름의 특수 값.

*PyObject** **PyStructSequence_New** (*PyObject* *type)

Return value: New reference. `PyStructSequence_NewType()`으로 만든 *type*의 인스턴스를 만듭니다.

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

Return value: Borrowed reference. *p*가 가리키는 구조체 시퀀스의 위치 *pos*에 있는 객체를 돌려줍니다. 범위 검사가 수행되지 않습니다.

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

Return value: Borrowed reference. `PyStructSequence_GetItem()`과 동등한 매크로.

void **PyStructSequence_SetItem** (*PyObject* *p, Py_ssize_t pos, *PyObject* *o)

구조체 시퀀스 *p*의 인덱스 *pos*에 있는 필드를 값 *o*로 설정합니다. `PyTuple_SET_ITEM()`과 마찬가지로, 이것은 새로운 인스턴스를 채울 때만 사용해야 합니다.

참고: 이 함수는 *o*에 대한 참조를 “훔칩니다”.

void **PyStructSequence_SET_ITEM** (*PyObject* *p, Py_ssize_t *pos, *PyObject* *o)
PyStructSequence_SetItem() 과 동등한 매크로.

참고: 이 함수는 *o*에 대한 참조를 “훔칩니다”.

8.3.6 리스트 객체

PyListObject

이 *PyObject*의 서브 형은 파이썬 리스트 객체를 나타냅니다.

PyTypeObject **PyList_Type**

이 *PyTypeObject* 인스턴스는 파이썬 리스트 형을 나타냅니다. 이것은 파이썬 계층의 `list`와 같은 객체입니다.

int **PyList_Check** (*PyObject* *p)

*p*가 리스트 객체나 리스트 형의 서브 형 인스턴스면 참을 반환합니다.

int **PyList_CheckExact** (*PyObject* *p)

*p*가 리스트 객체이지만 리스트 형의 서브 형의 인스턴스가 아니면 참을 반환합니다.

*PyObject** **PyList_New** (Py_ssize_t len)

Return value: New reference. 성공하면 길이 *len* 인 새 리스트를, 실패하면 NULL을 반환합니다.

참고: *len*이 0보다 크면, 반환된 리스트 객체의 항목은 NULL로 설정됩니다. 따라서 모든 항목을 *PyList_SetItem()*로 실제 객체로 설정하기 전에 *PySequence_SetItem()*와 같은 추상 API 함수를 사용하거나 파이썬 코드에 객체를 노출할 수 없습니다.

Py_ssize_t **PyList_Size** (*PyObject* *list)

*list*에서 리스트 객체의 길이를 반환합니다; 이는 리스트 객체에 대한 `len(list)`와 동등합니다.

Py_ssize_t **PyList_GET_SIZE** (*PyObject* *list)

에러 검사 없는 *PyList_Size()*의 매크로 형식.

*PyObject** **PyList_GetItem** (*PyObject* *list, Py_ssize_t index)

Return value: Borrowed reference. *list*가 가리키는 리스트에서 *index* 위치의 객체를 반환합니다. 위치는 음수가 아니어야 합니다; 리스트의 끝에서부터의 인덱싱은 지원되지 않습니다. *index*가 범위를 벗어나면 (<0 또는 >=len(list)), NULL을 반환하고 `IndexError` 예외를 설정합니다.

*PyObject** **PyList_GET_ITEM** (*PyObject* *list, Py_ssize_t i)

Return value: Borrowed reference. 에러 검사 없는 *PyList_GetItem()*의 매크로 형식.

int **PyList_SetItem** (*PyObject* *list, Py_ssize_t index, *PyObject* *item)

리스트의 인덱스 *index*에 있는 항목을 *item*으로 설정합니다. 성공하면 0을 반환합니다. *index*가 범위를 벗어나면, -1을 반환하고 `IndexError` 예외를 설정합니다.

참고: 이 함수는 *item*에 대한 참조를 “훔치고” 영향을 받는 위치의 리스트에 이미 있는 항목에 대한 참조를 버립니다.

void **PyList_SET_ITEM** (*PyObject* *list, Py_ssize_t i, *PyObject* *o)

에러 검사 없는 *PyList_SetItem()*의 매크로 형식. 일반적으로 이전 내용이 없는 새 리스트를 채우는 데 사용됩니다.

참고: 이 매크로는 *item*에 대한 참조를 “훔치고”, *PyList_SetItem()*과는 달리 대체되는 항목에 대한 참조를 버리지 않습니다; *list*의 *i* 위치에 있는 참조는 누수를 일으킵니다.

int **PyList_Insert** (*PyObject* *list, Py_ssize_t index, *PyObject* *item)

항목 *item*을 리스트 *list*의 인덱스 *index* 앞에 삽입합니다. 성공하면 0을 반환합니다; 실패하면 -1을 반환하고 예외를 설정합니다. *list.insert(index, item)*에 해당합니다.

int **PyList_Append** (*PyObject* *list, *PyObject* *item)

리스트 *list*의 끝에 객체 *item*을 추가합니다. 성공하면 0을 반환합니다; 실패하면 -1을 반환하고 예외를 설정합니다. *list.append(item)*에 해당합니다.

*PyObject** **PyList_GetSlice** (*PyObject* *list, Py_ssize_t low, Py_ssize_t high)

Return value: New reference. *list*에서 *low*와 *high* 사이에있는 객체들을 포함하는 리스트를 반환합니다. 실패하면 NULL을 반환하고 예외를 설정합니다. *list[low:high]*에 해당합니다. 리스트 끝에서부터의 인덱싱은 지원되지 않습니다.

int **PyList_SetSlice** (*PyObject* *list, Py_ssize_t low, Py_ssize_t high, *PyObject* *itemlist)

*low*와 *high* 사이의 *list* 슬라이스를 *itemlist*의 내용으로 설정합니다. *list[low:high] = itemlist*에 해당합니다. *itemlist*는 NULL일 수 있는데, 빈 리스트의 대입을 나타냅니다 (슬라이스 삭제). 성공하면 0을, 실패하면 -1을 반환합니다. 리스트 끝에서부터의 인덱싱은 지원되지 않습니다.

int **PyList_Sort** (*PyObject* *list)

list 항목을 제자리에서 정렬합니다. 성공하면 0을, 실패하면 -1을 반환합니다. 이것은 *list.sort()*와 동등합니다.

int **PyList_Reverse** (*PyObject* *list)

*list*의 항목을 제자리에서 뒤집습니다. 성공하면 0을, 실패하면 -1을 반환합니다. 이것은 *list.reverse()*와 동등합니다.

*PyObject** **PyList_AsTuple** (*PyObject* *list)

Return value: New reference. *list*의 내용을 포함하는 새 튜플 객체를 반환합니다; *tuple(list)*와 동등합니다.

int **PyList_ClearFreeList** ()

자유 목록(free list)을 비웁니다. 해제된 항목의 총수를 반환합니다.

버전 3.3에 추가.

8.4 컨테이너 객체

8.4.1 딕셔너리 객체

PyDictObject

이 *PyObject*의 서브 형은 파이썬 딕셔너리 객체를 나타냅니다.

PyTypeObject **PyDict_Type**

이 *PyTypeObject* 인스턴스는 파이썬 딕셔너리 형을 나타냅니다. 이것은 파이썬 계층의 *dict*와 같은 객체입니다.

int **PyDict_Check** (*PyObject* *p)

*p*가 *dict* 객체이거나 *dict* 형의 서브 형의 인스턴스면 참을 반환합니다.

`int PyDict_CheckExact (PyObject *p)`

*p*가 dict 객체이지만, dict 형의 서브 형의 인스턴스는 아니면 참을 반환합니다.

`PyObject* PyDict_New ()`

Return value: New reference. 새로운 빈 딕셔너리를 반환하거나, 실패하면 NULL을 반환합니다.

`PyObject* PyDictProxy_New (PyObject *mapping)`

Return value: New reference. 읽기 전용 동작을 강제하는 매핑을 위한 `types.MappingProxyType` 객체를 반환합니다. 이것은 일반적으로 비 동적 클래스 형을 위한 딕셔너리의 수정을 방지하기 위해 뷰를 만드는 데 사용됩니다.

`void PyDict_Clear (PyObject *p)`

기존 딕셔너리의 모든 키-값 쌍을 비웁니다.

`int PyDict_Contains (PyObject *p, PyObject *key)`

딕셔너리 *p*에 *key*가 포함되어 있는지 확인합니다. *p*의 항목이 *key*와 일치하면 1을 반환하고, 그렇지 않으면 0을 반환합니다. 에러면 -1을 반환합니다. 이는 파이썬 표현식 `key in p`와 동등합니다.

`PyObject* PyDict_Copy (PyObject *p)`

Return value: New reference. *p*와 같은 키-값 쌍을 포함하는 새 딕셔너리를 반환합니다.

`int PyDict_SetItem (PyObject *p, PyObject *key, PyObject *val)`

딕셔너리 *p*에 *val*을 *key* 키로 삽입합니다. *key*는 해시 가능해야 합니다. 그렇지 않으면 `TypeError`가 발생합니다. 성공하면 0을, 실패하면 -1을 반환합니다. 이 함수는 *val*에 대한 참조를 훔치지 않습니다.

`int PyDict_SetItemString (PyObject *p, const char *key, PyObject *val)`

*key*를 키로 사용하여 딕셔너리 *p*에 *val*을 삽입합니다. *key*는 `const char*`여야 합니다. 키 객체는 `PyUnicode_FromString(key)`를 사용하여 만듭니다. 성공하면 0을, 실패하면 -1을 반환합니다. 이 함수는 *val*에 대한 참조를 훔치지 않습니다.

`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. 딕셔너리 *p*에서 키가 *key*인 객체를 반환합니다. *key* 키가 없으면 예외를 설정하지 않고 NULL을 반환합니다.

`__hash__()`와 `__eq__()` 메서드를 호출하는 동안 발생하는 예외는 억제됩니다. 예러 보고를 얻으려면 대신 `PyDict_GetItemWithError()`를 사용하십시오.

`PyObject* PyDict_GetItemWithError (PyObject *p, PyObject *key)`

Return value: Borrowed reference. 예외를 억제하지 않는 `PyDict_GetItem()`의 변형입니다. 예외가 발생하면 예외를 설정하고 NULL을 반환합니다. 키가 없으면 예외를 설정하지 않고 NULL을 반환합니다.

`PyObject* PyDict_GetItemString (PyObject *p, const char *key)`

Return value: Borrowed reference. 이것은 `PyDict_GetItem()`와 같지만, *key*가 `PyObject*`가 아닌 `const char*`로 지정됩니다.

`__hash__()`와 `__eq__()` 메서드를 호출하고 임시 문자열 객체를 만드는 동안 발생하는 예외는 억제됩니다. 예러 보고를 얻으려면 대신 `PyDict_GetItemWithError()`를 사용하십시오.

`PyObject* PyDict_SetDefault (PyObject *p, PyObject *key, PyObject *defaultobj)`

Return value: Borrowed reference. 이것은 파이썬 수준의 `dict.setdefault()`와 같습니다. 존재하면, 딕셔너리 *p*에서 *key*에 해당하는 값을 반환합니다. 키가 dict에 없으면, 값 *defaultobj*로 삽입되고, *defaultobj*가 반환됩니다. 이 함수는 *key*의 해시 함수를 조회 및 삽입을 위해 독립적으로 평가하는 대신 한 번만 평가합니다.

버전 3.4에 추가.

*PyObject** **PyDict_Items** (*PyObject* *p)

Return value: New reference. 딕셔너리의 모든 항목을 포함하는 *PyListObject*를 반환합니다.

*PyObject** **PyDict_Keys** (*PyObject* *p)

Return value: New reference. 딕셔너리의 모든 키를 포함하는 *PyListObject*를 반환합니다.

*PyObject** **PyDict_Values** (*PyObject* *p)

Return value: New reference. 딕셔너리 *p*의 모든 값을 포함하는 *PyListObject*를 반환합니다.

Py_ssize_t **PyDict_Size** (*PyObject* *p)

딕셔너리에 있는 항목의 수를 반환합니다. 이는 딕셔너리에 대한 `len(p)`와 동등합니다.

int **PyDict_Next** (*PyObject* *p, *Py_ssize_t* *ppos, *PyObject* **pkey, *PyObject* **pvalue)

딕셔너리 *p*의 모든 키-값 쌍을 이터레이트 합니다. *ppos*에 의해 참조된 *Py_ssize_t*는, 이터레이션을 시작하기 위해 이 함수를 처음 호출하기 전에 0으로 초기화되어야 합니다; 이 함수는 딕셔너리의 각 쌍에 대해 참을 반환하고, 모든 쌍이 보고되었으면 거짓을 반환합니다. 매개 변수 *pkey*와 *pvalue*는 각각 키와 값으로 채울 *PyObject** 변수를 가리 키거나, NULL 일 수 있습니다. 이들을 통해 반환된 참조는 모두 빌린(borrowed) 것입니다. 이터레이션 중에 *ppos*를 변경하면 안 됩니다. 이 값은 내부 딕셔너리 구조 내의 오프셋을 나타내며, 구조가 희박하므로 오프셋이 연속되지 않습니다.

예를 들면:

```
PyObject *key, *value;
Py_ssize_t pos = 0;

while (PyDict_Next(self->dict, &pos, &key, &value)) {
    /* do something interesting with the values... */
    ...
}
```

딕셔너리 *p*는 이터레이션 중에 변경해서는 안 됩니다. 딕셔너리를 이터레이트 할 때 값을 변경하는 것은 안전하지만, 키 집합이 변경되지 않는 한만 그렇습니다. 예를 들면:

```
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)

매핑 객체 *b*를 이터레이트 하면서, 키-값 쌍을 딕셔너리 *a*에 추가합니다. *b*는 딕셔너리거나 *PyMapping_Keys()*와 *PyObject_GetItem()*를 지원하는 모든 객체일 수 있습니다. *override*가 참이면, *a*에 있는 기존 쌍이 *b*에서 일치하는 키가 있으면 교체되고, 그렇지 않으면 *a*와 일치하는 키가 없을 때만 쌍이 추가됩니다. 성공하면 0을 반환하고, 예외가 발생하면 -1을 반환합니다.

`int PyDict_Update(PyObject *a, PyObject *b)`

이는 C에서 `PyDict_Merge(a, b, 1)`와 같고, 두 번째 인자에 “keys” 어트리뷰트가 없을 때 `PyDict_Update()`가 키-값 쌍의 시퀀스에 대해 이터레이트 하지 않는다는 점만 제외하면, 파이썬에서 `a.update(b)`와 유사합니다. 성공하면 0을 반환하고, 예외가 발생하면 -1을 반환합니다.

`int PyDict_MergeFromSeq2(PyObject *a, PyObject *seq2, int override)`

`seq2`의 키-값 쌍으로 딕셔너리 `a`를 갱신하거나 병합합니다. `seq2`는 키-값 쌍으로 간주하는 길이 2의 이터러블 객체를 생성하는 이터러블 객체여야 합니다. 중복 키가 있으면, `override`가 참이면 마지막이 승리하고, 그렇지 않으면 첫 번째가 승리합니다. 성공 시 0을 반환하고, 예외가 발생하면 -1을 반환합니다. 동등한 파이썬은 이렇습니다(반환 값 제외)

```
def PyDict_MergeFromSeq2(a, seq2, override):
    for key, value in seq2:
        if override or key not in a:
            a[key] = value
```

`int PyDict_ClearFreeList()`

자유 목록(free list)을 비웁니다. 해제된 항목의 총수를 반환합니다.

버전 3.3에 추가.

8.4.2 집합 객체

이 절에서는 `set`과 `frozenset` 객체에 대한 공용 API에 대해 자세히 설명합니다. 아래 나열되지 않은 기능은 추상 객체 프로토콜 (`PyObject_CallMethod()`, `PyObject_RichCompareBool()`, `PyObject_Hash()`, `PyObject_Repr()`, `PyObject_IsTrue()`, `PyObject_Print()` 및 `PyObject_GetIter()`를 포함합니다) 이나 추상 숫자 프로토콜 (`PyNumber_And()`, `PyNumber_Subtract()`, `PyNumber_Or()`, `PyNumber_Xor()`, `PyNumber_InPlaceAnd()`, `PyNumber_InPlaceSubtract()`, `PyNumber_InPlaceOr()` 및 `PyNumber_InPlaceXor()`을 포함합니다).

PySetObject

이 `PyObject`의 서브 형은 `set`과 `frozenset` 객체 모두의 내부 데이터를 담는 데 사용됩니다. 이것은 작은 집합은 고정 크기(튜플 저장과 매우 흡사함)이고, 중형과 대형 집합은 별도의 가변 크기 메모리 블록(리스트 저장소처럼)을 가리킨다는 점에서 `PyDictObject`와 비슷합니다. 이 구조체의 필드는 아무것도 공개되지 않은 것으로 취급되어야 하며, 변경될 수 있습니다. 모든 액세스는 구조체의 값을 조작하기보다는 설명된 API를 통해 수행해야 합니다.

PyTypeObject PySet_Type

이것은 파이썬 `set` 형을 나타내는 `PyTypeObject`의 인스턴스입니다.

PyTypeObject PyFrozenSet_Type

이것은 파이썬 `frozenset` 형을 나타내는 `PyTypeObject`의 인스턴스입니다.

다음 형 검사 매크로는 모든 파이썬 객체에 대한 포인터에서 작동합니다. 마찬가지로, 생성자 함수는 모든 이터러블 파이썬 객체에서 작동합니다.

`int PySet_Check(PyObject *p)`

`p`가 `set` 객체나 서브 형의 인스턴스면 참을 반환합니다.

`int PyFrozenSet_Check(PyObject *p)`

`p`가 `frozenset` 객체나 서브 형의 인스턴스면 참을 반환합니다.

`int PyAnySet_Check(PyObject *p)`

`p`가 `set` 객체, `frozenset` 객체 또는 서브 형의 인스턴스면 참을 반환합니다.

`int PyAnySet_CheckExact(PyObject *p)`

`p`가 `set` 객체나 `frozenset` 객체이지만, 서브 형의 인스턴스는 아니면 참을 반환합니다.

`int PyFrozenSet_CheckExact (PyObject *p)`

*p*가 frozenset 객체이지만, 서브 형의 인스턴스는 아니면 참을 반환합니다.

*PyObject** `PySet_New (PyObject *iterable)`

Return value: New reference. *iterable*에 의해 반환된 객체를 포함하는 새로운 set을 반환합니다. *iterable*은 새로운 빈 집합을 만들기 위해 NULL 일 수 있습니다. 성공하면 새 집합을, 실패하면 NULL을 반환합니다. *iterable*이 실제로 이터러블이 아니면 `TypeError`를 발생시킵니다. 생성자는 집합을 복사할 때도 유용합니다 (`c=set(s)`).

*PyObject** `PyFrozenSet_New (PyObject *iterable)`

Return value: New reference. *iterable*에 의해 반환된 객체를 포함한 새로운 frozenset을 반환합니다. *iterable*은 새로운 빈 frozenset을 만들기 위해 NULL 일 수 있습니다. 성공하면 새 집합을, 실패하면 NULL을 반환합니다. *iterable*이 실제로 이터러블이 아니면 `TypeError`를 발생시킵니다.

set 이나 frozenset의 인스턴스 또는 그들의 서브 형의 인스턴스에 대해 다음 함수와 매크로를 사용할 수 있습니다.

`Py_ssize_t PySet_Size (PyObject *anyset)`

set 이나 frozenset 객체의 길이를 반환합니다. `len(anyset)`와 동등합니다. *anyset*이 set, frozenset 또는 서브 형의 인스턴스가 아니면 `PyExc_SystemError`를 발생시킵니다.

`Py_ssize_t PySet_GET_SIZE (PyObject *anyset)`

예러 검사 없는 `PySet_Size()`의 매크로 형식.

`int PySet_Contains (PyObject *anyset, PyObject *key)`

발견되면 1을, 발견되지 않으면 0을, 예러가 발생하면 -1을 반환합니다. 파이썬 `__contains__()` 메서드와는 달리, 이 함수는 해시 불가능한 집합을 임시 frozenset으로 자동 변환하지 않습니다. *key*가 해시 불가능하면, `TypeError`를 발생시킵니다. *anyset*이 set, frozenset 또는 서브 형의 인스턴스가 아니면 `PyExc_SystemError`를 발생시킵니다.

`int PySet_Add (PyObject *set, PyObject *key)`

*key*를 set 인스턴스에 추가합니다. 또한 frozenset 인스턴스에도 작동합니다 (`PyTuple_SetItem()`처럼 다른 코드에 노출되기 전에 새로운 frozenset의 값을 채우는 데 사용할 수 있습니다). 성공하면 0을, 실패하면 -1을 반환합니다. *key*가 해시 불가능하면, `TypeError`를 발생시킵니다. 성장할 공간이 없다면 `MemoryError`를 발생시킵니다. *set*이 set 이나 그 서브 형의 인스턴스가 아니면 `SystemError`를 발생시킵니다.

다음 함수는 set 이나 그것의 서브 형의 인스턴스에는 사용할 수 있지만, frozenset 이나 그 서브 형의 인스턴스에는 사용할 수 없습니다.

`int PySet_Discard (PyObject *set, PyObject *key)`

발견되고 제거되면 1을 반환하고, 발견되지 않으면(아무런 일도 하지 않습니다) 0을 반환하고, 예러가 발생하면 -1을 반환합니다. 발견할 수 없는 키에 대해 `KeyError`를 발생시키지 않습니다. *key*가 해시 불가능하면 `TypeError`를 발생시킵니다. 파이썬 `discard()` 메서드와는 달리, 이 함수는 해시 불가능한 집합을 임시 frozenset으로 자동 변환하지 않습니다. *set*이 set 이나 그 서브 형의 인스턴스가 아니면 `PyExc_SystemError`를 발생시킵니다.

*PyObject** `PySet_Pop (PyObject *set)`

Return value: New reference. *set*에 들어있는 임의의 객체에 대한 새 참조를 반환하고, *set*에서 객체를 제거합니다. 실패하면 NULL을 반환합니다. 집합이 비어 있으면, `KeyError`를 발생시킵니다. *set*이 set 이나 그 서브 형의 인스턴스가 아니면 `SystemError`를 발생시킵니다.

`int PySet_Clear (PyObject *set)`

기존의 모든 요소 집합을 비웁니다.

`int PySet_ClearFreeList ()`

자유 목록(free list)을 비웁니다. 해제된 항목의 총수를 반환합니다.

버전 3.3에 추가.

8.5 함수 객체

8.5.1 함수 객체

파이썬 함수와 관련된 몇 가지 함수가 있습니다.

PyFunctionObject

함수에 사용되는 C 구조체.

PyObject **PyFunction_Type**

이것은 *PyObject*의 인스턴스이며 파이썬 함수 형을 나타냅니다. 파이썬 프로그래머에게 `types.FunctionType`으로 노출됩니다.

int PyFunction_Check (*PyObject* *o)

*o*가 함수 객체(*PyFunction_Type* 형)면 참을 반환합니다. 매개 변수는 NULL이 아니어야 합니다.

*PyObject** **PyFunction_New** (*PyObject* *code, *PyObject* *globals)

Return value: New reference. 코드 객체 *code*와 연관된 새 함수 객체를 반환합니다. *globals*는 함수에서 액세스할 수 있는 전역 변수가 있는 딕셔너리이어야 합니다.

함수의 독스트링과 이름은 코드 객체에서 가져옵니다. `__module__`은 *globals*에서 가져옵니다. 인자 기본값, 어노테이션 및 클로저는 NULL로 설정됩니다. `__qualname__`은 함수의 이름과 같은 값으로 설정됩니다.

*PyObject** **PyFunction_NewWithQualName** (*PyObject* *code, *PyObject* *globals, *PyObject* *qualname)

Return value: New reference. *PyFunction_New()*와 비슷하지만, 함수 객체의 `__qualname__` 어트리뷰트를 설정할 수도 있도록 합니다. *qualname*은 유니코드 객체나 NULL이어야 합니다; NULL이면, `__qualname__` 어트리뷰트는 `__name__` 어트리뷰트와 같은 값으로 설정됩니다.

버전 3.3에 추가.

*PyObject** **PyFunction_GetCode** (*PyObject* *op)

Return value: Borrowed reference. 함수 객체 *op*와 연관된 코드 객체를 반환합니다.

*PyObject** **PyFunction_GetGlobals** (*PyObject* *op)

Return value: Borrowed reference. 함수 객체 *op*와 연관된 전역 딕셔너리를 반환합니다.

*PyObject** **PyFunction_GetModule** (*PyObject* *op)

Return value: Borrowed reference. 함수 객체 *op*의 `__module__` 어트리뷰트를 반환합니다. 이것은 일반적으로 모듈 이름을 포함하는 문자열이지만, 파이썬 코드로 다른 객체로 설정할 수 있습니다.

*PyObject** **PyFunction_GetDefaults** (*PyObject* *op)

Return value: Borrowed reference. 함수 객체 *op*의 인자 기본값을 반환합니다. 이는 인자의 튜플이나 NULL일 수 있습니다.

int PyFunction_SetDefaults (*PyObject* *op, *PyObject* *defaults)

함수 객체 *op*의 인자 기본값을 설정합니다. *defaults*는 `Py_None` 이나 튜플이어야 합니다.

실패하면 `SystemError`를 발생시키고 -1을 반환합니다.

*PyObject** **PyFunction_GetClosure** (*PyObject* *op)

Return value: Borrowed reference. 함수 객체 *op*와 연관된 클로저를 반환합니다. 이것은 NULL 이나 셀 객체의 튜플일 수 있습니다.

int PyFunction_SetClosure (*PyObject* *op, *PyObject* *closure)

함수 객체 *op*와 연관된 클로저를 설정합니다. *closure*는 `Py_None` 이나 셀 객체의 튜플이어야 합니다.

실패하면 `SystemError`를 발생시키고 -1을 반환합니다.

*PyObject** **PyFunction_GetAnnotations** (*PyObject** *op*)

Return value: Borrowed reference. 함수 객체 *op*의 어노테이션을 반환합니다. 이것은 가변 딕셔너리나 NULL일 수 있습니다.

int **PyFunction_SetAnnotations** (*PyObject** *op*, *PyObject** *annotations*)

함수 객체 *op*의 어노테이션을 설정합니다. *annotations*은 딕셔너리나 `Py_None` 이어야 합니다.

실패하면 `SystemError`를 발생시키고 -1을 반환합니다.

8.5.2 인스턴스 메서드 객체

인스턴스 메서드는 `PyCFunction`에 대한 래퍼이며 `PyCFunction`를 클래스 객체에 연결하는 새로운 방법입니다. 이전의 `PyMethod_New(func, NULL, class)` 호출을 대체합니다.

PyObject **PyInstanceMethod_Type**

이 *PyObject* 인스턴스는 파이썬 인스턴스 메서드 형을 나타냅니다. 파이썬 프로그램에는 노출되지 않습니다.

int **PyInstanceMethod_Check** (*PyObject** *o*)

*o*가 인스턴스 메서드 객체면 참을 반환합니다 (*PyInstanceMethod_Type* 형입니다). 매개 변수는 NULL이 아니어야 합니다.

*PyObject** **PyInstanceMethod_New** (*PyObject** *func*)

Return value: New reference. 새 인스턴스 메서드 객체를 반환합니다. *func*는 임의의 콜러블 객체인데, *func*는 인스턴스 메서드가 호출될 때 호출될 함수입니다.

*PyObject** **PyInstanceMethod_Function** (*PyObject** *im*)

Return value: Borrowed reference. 인스턴스 메서드 *im*과 연관된 함수 객체를 반환합니다.

*PyObject** **PyInstanceMethod_GET_FUNCTION** (*PyObject** *im*)

Return value: Borrowed reference. 오류 검사를 피하는 `PyInstanceMethod_Function()`의 매크로 버전.

8.5.3 메서드 객체

메서드는 연결된 (bound) 함수 객체입니다. 메서드는 항상 사용자 정의 클래스의 인스턴스에 연결됩니다. 연결되지 않은 (unbound) 메서드(클래스 객체에 연결된 메서드)는 더는 사용할 수 없습니다.

PyObject **PyMethod_Type**

이 *PyObject* 인스턴스는 파이썬 메서드 형을 나타냅니다. 이것은 파이썬 프로그램에 `types.MethodType`로 노출됩니다.

int **PyMethod_Check** (*PyObject** *o*)

*o*가 메서드 객체면 참을 반환합니다 (*PyMethod_Type* 형입니다). 매개 변수는 NULL이 아니어야 합니다.

*PyObject** **PyMethod_New** (*PyObject** *func*, *PyObject** *self*)

Return value: New reference. 새로운 메서드 객체를 돌려줍니다. *func*는 임의의 콜러블 객체이며, *self*는 메서드가 연결되어야 할 인스턴스입니다. *func*는 메서드가 호출될 때 호출될 함수입니다. *self*는 NULL이 아니어야 합니다.

*PyObject** **PyMethod_Function** (*PyObject** *meth*)

Return value: Borrowed reference. *meth* 메서드와 연관된 함수 객체를 반환합니다.

*PyObject** **PyMethod_GET_FUNCTION** (*PyObject** *meth*)

Return value: Borrowed reference. 오류 검사를 피하는 `PyMethod_Function()`의 매크로 버전.

*PyObject** **PyMethod_Self** (*PyObject** *meth*)

Return value: Borrowed reference. *meth* 메서드와 연관된 인스턴스를 반환합니다.

*PyObject** **PyMethod_GET_SELF** (*PyObject* *meth)

Return value: Borrowed reference. 오류 검사를 피하는 *PyMethod_Self()*의 매크로 버전.

int **PyMethod_ClearFreeList** ()

자유 목록을 지웁니다. 해제된 총 항목 수를 반환합니다.

8.5.4 셀 객체

“셀” 객체는 여러 스코프에서 참조하는 변수를 구현하는 데 사용됩니다. 이러한 변수마다, 값을 저장하기 위해 셀 객체가 만들어집니다; 값을 참조하는 각 스택 프레임의 지역 변수에는 해당 변수를 사용하는 외부 스코프의 셀에 대한 참조가 포함됩니다. 값에 액세스하면, 셀 객체 자체 대신 셀에 포함된 값이 사용됩니다. 이러한 셀 객체의 역참조(de-referencing)는 생성된 바이트 코드로부터의 지원이 필요합니다; 액세스 시 자동으로 역참조 되지 않습니다. 셀 객체는 다른 곳에 유용하지는 않습니다.

PyCellObject

셀 객체에 사용되는 C 구조체.

PyTypeObject **PyCell_Type**

셀 객체에 해당하는 형 객체.

int **PyCell_Check** (ob)

ob가 셀 객체면 참을 반환합니다; ob는 NULL이 아니어야 합니다.

*PyObject** **PyCell_New** (*PyObject* *ob)

Return value: New reference. ob 값을 포함하는 새 셀 객체를 만들고 반환합니다. 매개 변수는 NULL 일 수 있습니다.

*PyObject** **PyCell_Get** (*PyObject* *cell)

Return value: New reference. 셀 cell의 내용을 반환합니다.

*PyObject** **PyCell_GET** (*PyObject* *cell)

Return value: Borrowed reference. 셀 cell의 내용을 반환하지만, cell이 NULL이 아닌지와 셀 객체인지를 확인하지 않습니다.

int **PyCell_Set** (*PyObject* *cell, *PyObject* *value)

셀 객체 cell의 내용을 value로 설정합니다. 이렇게 하면 셀의 현재 내용에 대한 참조를 해제합니다. value는 NULL 일 수 있습니다. cell는 NULL이 아니어야 합니다; 셀 객체가 아니면, -1이 반환됩니다. 성공하면, 0이 반환됩니다.

void **PyCell_SET** (*PyObject* *cell, *PyObject* *value)

셀 객체 cell의 값을 value로 설정합니다. 참조 횟수는 조정되지 않고, 안전을 위한 검사가 이루어지지 않습니다; cell은 NULL이 아니어야 하고 셀 객체여야 합니다.

8.5.5 코드 객체

코드 객체는 CPython 구현의 저수준 세부 사항입니다. 각 객체는 아직 함수에 묶여 있지 않은 실행 가능한 코드 덩어리를 나타냅니다.

PyCodeObject

코드 객체를 설명하는 데 사용되는 객체의 C 구조체. 이 형의 필드는 언제든지 변경될 수 있습니다.

PyTypeObject **PyCode_Type**

이것은 Python code 형을 나타내는 *PyTypeObject*의 인스턴스입니다.

int **PyCode_Check** (*PyObject* *co)

co가 code 객체면 참을 반환합니다.

int **PyCode_GetNumFree** (*PyCodeObject* *co)

co에 있는 자유 변수의 개수를 반환합니다.

*PyCodeObject** **PyCode_New** (int *argcount*, int *kwonlyargcount*, int *nlocals*, int *stacksize*, int *flags*, *PyObject* **code*, *PyObject* **consts*, *PyObject* **names*, *PyObject* **varnames*, *PyObject* **freevars*, *PyObject* **cellvars*, *PyObject* **filename*, *PyObject* **name*, int *firstlineno*, *PyObject* **notab*)

Return value: New reference. 새 코드 객체를 반환합니다. 프레임 만들기 위해 더미 코드 객체가 필요하면, 대신 *PyCode_NewEmpty()* 를 사용하십시오. 바이트 코드의 정의가 자주 변경되기 때문에, *PyCode_New()* 를 직접 호출하면 정확한 파이썬 버전에 구속될 수 있습니다.

*PyCodeObject** **PyCode_NewWithPosOnlyArgs** (int *argcount*, int *posonlyargcount*, int *kwonlyargcount*, int *nlocals*, int *stacksize*, int *flags*, *PyObject* **code*, *PyObject* **consts*, *PyObject* **names*, *PyObject* **varnames*, *PyObject* **freevars*, *PyObject* **cellvars*, *PyObject* **filename*, *PyObject* **name*, int *firstlineno*, *PyObject* **notab*)

Return value: New reference. *PyCode_New()* 와 비슷하지만, 위치 전용 인자를 위한 추가 “posonlyargcount” 가 있습니다.

버전 3.8에 추가.

*PyCodeObject** **PyCode_NewEmpty** (const char **filename*, const char **funcname*, int *firstlineno*)

Return value: New reference. 지정된 파일명, 함수명 및 첫 번째 줄 번호를 갖는 새 빈 코드 객체를 반환합니다. 결과 코드 객체를 *exec()* 또는 *eval()* 하는 것은 불법입니다.

8.6 기타 객체

8.6.1 파일 객체

이 API는 C 표준 라이브러리의 버퍼링 된 I/O (FILE*) 지원에 의존하는 내장 파일 객체에 대한 파이썬 2 C API의 최소 예시입니다. 파이썬 3에서, 파일과 스트림은 새로운 io 모듈을 사용합니다. 이 모듈은 운영체제의 저수준 버퍼링 되지 않은 I/O 위에 여러 계층을 정의합니다. 아래에서 설명하는 함수는 이러한 새로운 API에 대한 편리한 C 래퍼이며, 주로 인터프리터의 내부 오류 보고를 위한 것입니다; 제삼자 코드는 대신 io API에 액세스하는 것이 좋습니다.

*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. 이미 열려있는 파일의 파일 기술자 *fd*로 파이썬 파일 객체를 만듭니다. 인자 *name*, *encoding*, *errors* 및 *newline*은 기본값을 사용하기 위해 NULL 일 수 있습니다; *buffering*은 기본값을 사용하기 위해 -1 일 수 있습니다. *name*은 무시되고, 이전 버전과의 호환성을 위해 유지됩니다. 실패 시 NULL을 반환합니다. 인자에 대한 더 자세한 설명은 *io.open()* 함수 설명서를 참조하십시오.

경고: 파이썬 스트림이 자체적인 버퍼링 계층을 가지고 있으므로, OS 수준의 파일 기술자와 혼합하면 여러 예기치 못한 문제가 발생할 수 있습니다(가령 데이터의 예상치 못한 순서).

버전 3.2에서 변경: *name* 어트리뷰트를 무시합니다.

int **PyObject_AsFileDescriptor** (*PyObject* **p*)

*p*와 관련된 파일 기술자를 int로 반환합니다. 객체가 정수면, 값이 반환됩니다. 그렇지 않으면 객체의 *fileno()* 메서드가 있으면 호출됩니다; 메서드는 반드시 정수를 반환해야 하고, 그 값이 파일 기술자 값으로 반환됩니다. 실패하면 예외를 설정하고 -1을 반환합니다.

*PyObject** **PyFile_GetLine** (*PyObject* **p*, int *n*)

Return value: New reference. *p.readline()* 과 동등합니다. 이 함수는 객체 *p*에서 한 줄을 읽습니다. *p*는 파일 객체나 *readline()* 메서드가 있는 임의의 객체일 수 있습니다. *n*이 0이면, 줄의 길이와 관계 없이 정확히 한 줄을 읽습니다. *n*이 0보다 크면, *n* 바이트 이상을 파일에서 읽지 않습니다; 불완전한 줄이

반환될 수 있습니다. 두 경우 모두, 파일 끝에 즉시 도달하면 빈 문자열이 반환됩니다. 그러나 n 이 0보다 작으면, 길이와 관계없이 한 줄을 읽지만, 파일 끝에 즉시 도달하면 EOFError가 발생합니다.

int PyFile_SetOpenCodeHook (Py_OpenCodeHookFunction *handler*)

제공된 *handler*를 통해 매개 변수를 전달하도록 `io.open_code()`의 일반적인 동작을 재정의합니다.

*handler*는 형이 `PyObject *(*)(PyObject *path, void *userData)`인 함수이며, 여기서 *path*는 `PyUnicodeObject`임이 보장됩니다.

userData 포인터는 *hook* 함수로 전달됩니다. *hook* 함수는 다른 런타임에서 호출될 수 있으므로, 이 포인터는 파이썬 상태를 직접 참조하면 안 됩니다.

이 *hook*은 의도적으로 임포트 중에 사용되므로, 고정되었거나(frozen) `sys.modules`에 있다고 알려진 경우가 아니라면 *hook* 실행 중에 새로운 모듈을 임포트하는 것을 피하십시오.

일단 *hook*이 설정되면, 제거하거나 교체할 수 없으며, 이후의 `PyFile_SetOpenCodeHook()`에 대한 호출은 실패합니다. 실패 시, 함수는 -1을 반환하고 인터프리터가 초기화되었으면 예외를 설정합니다.

이 함수는 `Py_Initialize()` 전에 호출해도 안전합니다.

Raises an auditing event `setopencodehook` with no arguments.

버전 3.8에 추가.

int PyFile_WriteObject (PyObject **obj*, PyObject **p*, int *flags*)

객체 *obj*를 파일 객체 *p*에 씁니다. *flags*에서 지원되는 유일한 플래그는 `Py_PRINT_RAW`입니다; 주어진다면, `repr()` 대신 객체의 `str()`이 기록됩니다. 성공하면 0을, 실패하면 -1을 반환합니다; 적절한 예외가 설정됩니다.

int PyFile_WriteString (const char **s*, PyObject **p*)

문자열 *s*를 파일 객체 *p*에 씁니다. 성공하면 0을 반환하고, 실패하면 -1을 반환합니다; 적절한 예외가 설정됩니다.

8.6.2 모듈 객체

PyTypeObject PyModule_Type

이 `PyTypeObject` 인스턴스는 파이썬 모듈 형을 나타냅니다. 이것은 `types.ModuleType`으로 파이썬 프로그램에 노출됩니다.

int PyModule_Check (PyObject **p*)

*p*가 모듈 객체이거나 모듈 객체의 서브 형이면 참을 반환합니다.

int PyModule_CheckExact (PyObject **p*)

*p*가 모듈 객체이지만, `PyModule_Type`의 서브 형이 아니면 참을 반환합니다.

PyObject* PyModule_NewObject (PyObject **name*)

Return value: New reference. `__name__` 어트리뷰트가 *name*으로 설정된 새 모듈 객체를 반환합니다. 모듈의 `__name__`, `__doc__`, `__package__` 및 `__loader__` 어트리뷰트가 채워집니다(`__name__`을 제외하고 모두 None으로 설정됩니다); `__file__` 어트리뷰트를 제공하는 것은 호출자의 책임입니다.

버전 3.3에 추가.

버전 3.4에서 변경: `__package__`와 `__loader__`가 None으로 설정됩니다.

PyObject* PyModule_New (const char **name*)

Return value: New reference. `PyModule_NewObject()`와 비슷하지만, *name*이 유니코드 객체 대신 UTF-8로 인코딩된 문자열입니다.

PyObject* PyModule_GetDict (PyObject **module*)

Return value: Borrowed reference. *module*의 이름 공간을 구현하는 딕셔너리 객체를 반환합니다; 이 객체는

모듈 객체의 `__dict__` 어트리뷰트와 같습니다. `module`이 모듈 객체(또는 모듈 객체의 서브 형)가 아니면, `SystemError`가 발생하고 `NULL`이 반환됩니다.

확장은 모듈의 `__dict__`를 직접 조작하지 말고 다른 `PyModule_*`()과 `PyObject_*`() 함수를 사용하는 것이 좋습니다.

*PyObject** **PyModule_GetNameObject** (*PyObject* *module)

Return value: New reference. module의 `__name__` 값을 반환합니다. 모듈이 제공하지 않거나, 문자열이 아니면, `SystemError`가 발생하고 `NULL`이 반환됩니다.

버전 3.3에 추가.

const char* **PyModule_GetName** (*PyObject* *module)

PyModule_GetNameObject()와 비슷하지만 'utf-8'로 인코딩된 이름을 반환합니다.

void* **PyModule_GetState** (*PyObject* *module)

모듈의 “상태”, 즉 모듈 생성 시 할당된 메모리 블록을 가리키는 포인터나 `NULL`을 반환합니다. *PyModuleDef.m_size*를 참조하십시오.

*PyModuleDef** **PyModule_GetDef** (*PyObject* *module)

모듈이 만들어진 *PyModuleDef* 구조체에 대한 포인터나 모듈이 정의에서 만들어지지 않았으면 `NULL`을 반환합니다.

*PyObject** **PyModule_GetFilenameObject** (*PyObject* *module)

Return value: New reference. module의 `__file__` 어트리뷰트를 사용하여 module이 로드된 파일 이름을 반환합니다. 정의되지 않았거나 유니코드 문자열이 아니면, `SystemError`를 발생시키고 `NULL`을 반환합니다; 그렇지 않으면 유니코드 객체에 대한 참조를 반환합니다.

버전 3.2에 추가.

const char* **PyModule_GetFilename** (*PyObject* *module)

PyModule_GetFilenameObject()와 비슷하지만 'utf-8'로 인코딩된 파일명을 반환합니다.

버전 3.2부터 폐지: *PyModule_GetFilename*()은 인코딩할 수 없는 파일명에 대해 `UnicodeEncodeError`를 발생시킵니다, 대신 *PyModule_GetFilenameObject*()를 사용하십시오.

C 모듈 초기화

모듈 객체는 일반적으로 확장 모듈(초기화 함수를 내보내는 공유 라이브러리)이나 컴파일된 모듈(초기화 함수가 *PyImport_AppendInittab*()을 사용하여 추가된)에서 만들어집니다. 자세한 내용은 *building*나 *extending-with-embedding*를 참조하십시오.

초기화 함수는 모듈 정의 인스턴스를 *PyModule_Create*()에 전달하고 결과 모듈 객체를 반환하거나, 정의 구조체 자체를 반환하여 “다단계 초기화”를 요청할 수 있습니다.

PyModuleDef

모듈 객체를 만드는 데 필요한 모든 정보를 담고 있는 모듈 정의 구조체. 일반적으로 각 모듈에 대해 이 형의 정적으로 초기화된 변수가 하나만 있습니다.

PyModuleDef_Base **m_base**

이 멤버를 항상 *PyModuleDef_HEAD_INIT*로 초기화하십시오.

const char ***m_name**

새 모듈의 이름.

const char ***m_doc**

모듈의 독스트링; 일반적으로 *PyDoc_STRVAR*로 만들어진 독스트링 변수가 사용됩니다.

Py_ssize_t m_size

모듈 상태는 정적 전역이 아닌 `PyModule_GetState()`로 조회할 수 있는 모듈별 메모리 영역에 유지될 수 있습니다. 이것은 여러 서브 인터프리터에서 모듈을 사용하는 것을 안전하게 만듭니다.

이 메모리 영역은 모듈 생성 시 `m_size`를 기준으로 할당되며, 모듈 객체가 할당 해제될 때 (있다면 `m_free` 함수가 호출된 후에) 해제됩니다.

`m_size`를 -1로 설정하면 모듈이 전역 상태를 갖기 때문에 서브 인터프리터를 지원하지 않는다는 뜻입니다.

음수가 아닌 값으로 설정하면 모듈을 다시 초기화 할 수 있다는 뜻이며 상태에 필요한 추가 메모리 양을 지정합니다. 단단계 초기화에는 음이 아닌 `m_size`가 필요합니다.

자세한 내용은 **PEP 3121**을 참조하십시오.

PyMethodDef* m_methods

`PyMethodDef` 값으로 기술되는 모듈 수준 함수 테이블에 대한 포인터. 함수가 없으면 NULL일 수 있습니다.

PyModuleDef_Slot* m_slots

단단계 초기화를 위한 슬롯 정의 배열, {0, NULL} 항목으로 종료됩니다. 단단계 초기화를 사용할 때, `m_slots`는 NULL이어야 합니다.

버전 3.5에서 변경: 버전 3.5 이전에는, 이 멤버가 항상 NULL로 설정되었으며, 다음과 같이 정의되었습니다:

`inquiry m_reload`

traverseproc m_traverse

모듈 객체의 GC 탐색 중 호출할 탐색 함수나, 필요하지 않으면 NULL. 이 함수는 모듈 상태가 할당되기 전에 (`PyModule_GetState()`가 NULL을 반환할 수 있습니다), 그리고 `Py_mod_exec` 함수가 실행되기 전에 호출될 수 있습니다.

inquiry m_clear

모듈 객체의 GC 정리 중에 호출할 정리(clear) 함수나, 필요하지 않으면 NULL. 이 함수는 모듈 상태가 할당되기 전에 (`PyModule_GetState()`가 NULL을 반환할 수 있습니다), 그리고 `Py_mod_exec` 함수가 실행되기 전에 호출될 수 있습니다.

freefunc m_free

모듈 객체 할당 해제 중에 호출할 함수나, 필요하지 않으면 NULL. 이 함수는 모듈 상태가 할당되기 전에 (`PyModule_GetState()`가 NULL을 반환할 수 있습니다), 그리고 `Py_mod_exec` 함수가 실행되기 전에 호출될 수 있습니다.

단단계 초기화

모듈 초기화 함수는 모듈 객체를 직접 만들고 반환할 수 있습니다. 이것을 “단단계 초기화”라고 하며, 다음 두 모듈 생성 함수 중 하나를 사용합니다:

PyObject* PyModule_Create (PyModuleDef *def)

Return value: New reference. `def`의 정의에 따라, 새 모듈 객체를 만듭니다. 이것은 `module_api_version`이 PYTHON_API_VERSION으로 설정된 `PyModule_Create2()`처럼 동작합니다.

PyObject* PyModule_Create2 (PyModuleDef *def, int module_api_version)

Return value: New reference. `def`의 정의에 따라, API 버전 `module_api_version`을 가정하여 새 모듈 객체를 만듭니다. 해당 버전이 실행 중인 인터프리터 버전과 일치하지 않으면, RuntimeWarning을 발생시킵니다.

참고: 이 함수는 대부분 `PyModule_Create()`를 대신 사용해야 합니다; 확실히 필요할 때만 사용하십시오.

초기화 함수에서 반환되기 전에, 결과 모듈 객체는 일반적으로 `PyModule_AddObject()`와 같은 함수를 사용하여 채워집니다.

다단계 초기화

확장을 지정하는 다른 방법은 “다단계 초기화”를 요청하는 것입니다. 이 방법으로 만들어진 확장 모듈은 파이썬 모듈과 더 비슷하게 동작합니다: 초기화는 모듈 객체가 만들어질 때의 생성 단계(*creation phase*)와 채워질 때의 실행 단계(*execution phase*)로 분할됩니다. 구별은 클래스의 `__new__()`와 `__init__()` 메서드와 유사합니다.

단단계 초기화를 사용하여 만들어진 모듈과 달리, 이 모듈은 싱글톤이 아닙니다: `sys.modules` 항목을 제거하고 모듈을 다시 임포트 하면, 새 모듈 객체가 만들어지고, 이전 모듈은 일반 가비지 수집이 적용됩니다 – 파이썬 모듈과 마찬가지로입니다. 기본적으로, 같은 정의에서 만들어진 여러 모듈은 독립적이어야 합니다: 하나를 변경해도 다른 모듈에는 영향을 미치지 않습니다. 즉, 모든 상태는 모듈 객체(예를 들어 `PyModule_GetState()`를 사용해서)나 그 내용(가령 모듈의 `__dict__`나 `PyType_FromSpec()`으로 만든 개별 클래스)으로 제한되어야 합니다.

다단계 초기화를 사용하여 만들어진 모든 모듈은 서브 인터프리터를 지원할 것으로 기대됩니다. 다중 모듈을 독립적으로 유지하는 것은 일반적으로 이를 달성하기에 충분합니다.

다단계 초기화를 요청하기 위해, 초기화 함수(`PyInit_modulename`)는 비어 있지 않은 `m_slots`를 가진 `PyModuleDef` 인스턴스를 반환합니다. 반환되기 전에, `PyModuleDef` 인스턴스를 다음 함수를 사용하여 초기화해야 합니다:

`PyObject* PyModuleDef_Init(PyModuleDef *def)`

Return value: Borrowed reference. 모듈 정의가 형과 참조 횟수를 올바르게 보고하는 올바르게 초기화된 파이썬 객체에게 합니다.

`def`를 `PyObject*`로 캐스트 하거나, 에러가 발생하면 `NULL`을 반환합니다.

버전 3.5에 추가.

모듈 정의의 `m_slots` 멤버는 `PyModuleDef_Slot` 구조체의 배열을 가리켜야 합니다:

`PyModuleDef_Slot`

`int slot`

아래 설명된 사용 가능한 값 중에서 선택된, 슬롯 ID.

`void* value`

슬롯 ID에 따라 그 의미가 달라지는, 슬롯의 값.

버전 3.5에 추가.

`m_slots` 배열은 id가 0인 슬롯으로 종료해야 합니다.

사용 가능한 슬롯 형은 다음과 같습니다:

`Py_mod_create`

모듈 객체 자체를 만들기 위해 호출되는 함수를 지정합니다. 이 슬롯의 `value` 포인터는 다음과 같은 서명을 갖는 함수를 가리켜야 합니다:

`PyObject* create_module(PyObject *spec, PyModuleDef *def)`

이 함수는 **PEP 451**에 정의된 대로, `ModuleSpec` 인스턴스와 모듈 정의를 받습니다. 새 모듈 객체를 반환하거나, 에러를 설정하고 `NULL`을 반환해야 합니다.

이 함수는 최소한으로 유지해야 합니다. 특히 같은 모듈을 다시 임포트 하려고 시도하면 무한 루프가 발생할 수 있어서, 임의의 파이썬 코드를 호출하면 안 됩니다.

하나의 모듈 정의에서 여러 `Py_mod_create` 슬롯을 지정할 수 없습니다.

`Py_mod_create`를 지정하지 않으면, 임포트 절차는 `PyModule_New()`를 사용하여 일반 모듈 객체를 만듭니다. 이름은 정의가 아니라 *spec*에서 취합니다, 확장 모듈이 단일 모듈 정의를 공유하면서 모듈 계층 구조에서 해당 위치에 동적으로 조정되고 심볼릭 링크를 통해 다른 이름으로 임포트 될 수 있도록 하기 위함입니다.

반환된 객체가 `PyModule_Type`의 인스턴스 일 필요는 없습니다. 임포트 관련 어트리뷰트 설정과 읽기를 지원하는 한 모든 형을 사용할 수 있습니다. 그러나, `PyModuleDef`에 `NULL`이 아닌 `m_traverse`, `m_clear`, `m_free`; 0이 아닌 `m_size`; 또는 `Py_mod_create` 이외의 슬롯이 있으면, `PyModule_Type` 인스턴스 만 반환될 수 있습니다.

Py_mod_exec

모듈을 실행하기 위해 호출되는 함수를 지정합니다. 이것은 파이썬 모듈의 코드를 실행하는 것과 동등합니다: 일반적으로, 이 함수는 클래스와 상수를 모듈에 추가합니다. 함수의 서명은 다음과 같습니다:

```
int exec_module (PyObject* module)
```

여러 개의 `Py_mod_exec` 슬롯이 지정되면, `m_slots` 배열에 나타나는 순서대로 처리됩니다.

다단계 초기화에 대한 자세한 내용은 [PEP 489](#)를 참조하십시오.

저수준 모듈 생성 함수

다단계 초기화를 사용할 때 수면 아래에서는 다음 함수가 호출됩니다. 이들은 직접 사용할 수 있는데, 예를 들어 모듈 객체를 동적으로 생성할 때 그렇습니다. 모듈을 완전히 초기화하려면 `PyModule_FromDefAndSpec`과 `PyModule_ExecDef`를 모두 호출해야 함에 유의하십시오.

PyObject * **PyModule_FromDefAndSpec** (*PyModuleDef* *def, *PyObject* *spec)

Return value: New reference. 주어진 모듈의 정의와 `ModuleSpec spec`으로 새 모듈 객체를 만듭니다. 이것은 `module_api_version`이 `PYTHON_API_VERSION`으로 설정된 `PyModule_FromDefAndSpec2()` 처럼 동작합니다.

버전 3.5에 추가.

PyObject * **PyModule_FromDefAndSpec2** (*PyModuleDef* *def, *PyObject* *spec, int module_api_version)

Return value: New reference. API 버전 `module_api_version`을 가정하여, 주어진 모듈의 정의와 `ModuleSpec spec`으로 새 모듈 객체를 만듭니다. 해당 버전이 실행 중인 인터프리터 버전과 일치하지 않으면, `RuntimeWarning`을 발생시킵니다.

참고: 이 함수는 대부분 `PyModule_FromDefAndSpec()`을 대신 사용해야 합니다; 확실히 필요할 때만 사용하십시오.

버전 3.5에 추가.

```
int PyModule_ExecDef (PyObject *module, PyModuleDef *def)
def에 지정된 모든 실행 슬롯(Py_mod_exec)을 처리합니다.
```

버전 3.5에 추가.

```
int PyModule_SetDocString (PyObject *module, const char *docstring)
```

`module`의 독스트링을 `docstring`으로 설정합니다. 이 함수는 `PyModule_Create`나 `PyModule_FromDefAndSpec`을 사용하여 `PyModuleDef`에서 모듈을 만들 때 자동으로 호출됩니다.

버전 3.5에 추가.

int PyModule_AddFunctions (*PyObject* *module, *PyMethodDef* *functions)

NULL 종료 *functions* 배열의 함수를 *module*에 추가합니다. 개별 항목에 대한 자세한 내용은 *PyMethodDef* 설명서를 참조하십시오 (공유 모듈 이름 공간이 없기 때문에, C로 구현된 모듈 수준 “함수(functions)”는 일반적으로 첫 번째 매개 변수로 모듈을 수신하여, 파이썬 클래스의 인스턴스 메서드와 유사하게 만듭니다). 이 함수는 *PyModule_Create*나 *PyModule_FromDefAndSpec*을 사용하여 *PyModuleDef*에서 모듈을 만들 때 자동으로 호출됩니다.

버전 3.5에 추가.

지원 함수

모듈 초기화 함수(단단계 초기화를 사용하는 경우)나 모듈 실행 슬롯에서 호출되는 함수(다단계 초기화를 사용하는 경우)는, 모듈 상태 초기화를 도우려고 다음 함수를 사용할 수 있습니다:

int PyModule_AddObject (*PyObject* *module, const char *name, *PyObject* *value)

*name*으로 *module*에 객체를 추가합니다. 모듈의 초기화 함수에서 사용할 수 있는 편의 함수입니다. 성공 시 *value*에 대한 참조를 훔칩니다. 에러 시 -1을, 성공하면 0을 반환합니다.

참고: 참조를 훔치는 다른 함수와 달리, *PyModule_AddObject* () 는 성공 시에만 *value*의 참조 횟수를 감소시킵니다.

이는 반환 값을 확인해야 하며, 에러 시 호출하는 코드가 수동으로 *value*를 *Py_DECREF* () 해야 함을 뜻합니다. 사용법 예:

```
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)

*module*에 정수 상수를 *name*으로 추가합니다. 이 편의 함수는 모듈의 초기화 함수에서 사용할 수 있습니다. 에러 시 -1을, 성공하면 0을 반환합니다.

int PyModule_AddStringConstant (*PyObject* *module, const char *name, const char *value)

*module*에 문자열 상수를 *name*으로 추가합니다. 이 편의 함수는 모듈의 초기화 함수에서 사용할 수 있습니다. 문자열 *value*는 NULL로 끝나야 합니다. 에러 시 -1을, 성공 시 0을 반환합니다.

int PyModule_AddIntMacro (*PyObject* *module, macro)

*module*에 int 상수를 추가합니다. 이름과 값은 *macro*에서 취합니다. 예를 들어 *PyModule_AddIntMacro*(module, AF_INET)은 *AF_INET* 값을 가진 int 상수 *AF_INET*을 *module*에 추가합니다. 에러 시 -1을, 성공하면 0을 반환합니다.

int PyModule_AddStringMacro (*PyObject* *module, macro)

*module*에 문자열 상수를 추가합니다.

모듈 조회

단단계 초기화는 현재 인터프리터의 컨텍스트에서 조회할 수 있는 싱글톤 모듈을 만듭니다. 이는 나중에 모듈 정의에 대한 참조만으로 모듈 객체를 검색할 수 있도록 합니다.

이 함수들은 단단계 초기화를 사용하여 만들어진 모듈에서는 작동하지 않습니다. 단일 정의에서 그러한 모듈이 여러 개 만들어질 수 있기 때문입니다.

*PyObject** **PyState_FindModule** (*PyModuleDef* *def)

Return value: Borrowed reference. 현재 인터프리터에 대해 def에서 만들어진 모듈 객체를 반환합니다. 이 메서드를 사용하려면 먼저 모듈 객체가 *PyState_AddModule()*로 인터프리터 상태에 연결되어 있어야 합니다. 해당 모듈 객체를 찾을 수 없거나 인터프리터 상태에 아직 연결되지 않았으면, NULL을 반환합니다.

int **PyState_AddModule** (*PyObject* *module, *PyModuleDef* *def)

함수에 전달된 모듈 객체를 인터프리터 상태에 연결합니다. 이는 *PyState_FindModule()*을 통해 모듈 객체에 액세스 할 수 있도록 합니다.

단단계 초기화를 사용하여 만든 모듈에만 효과가 있습니다.

파이썬은 모듈을 임포트 한 후 자동으로 *PyState_AddModule*을 호출하므로, 모듈 초기화 코드에서 호출하는 것은 불필요합니다 (하지만 무해합니다). 모듈의 자체 초기화 코드가 추후 *PyState_FindModule*을 호출하는 경우에만 명시적인 호출이 필요합니다. 이 함수는 주로 대안 임포트 메커니즘을 구현하기 위한 것입니다 (직접 호출하거나, 필요한 상태 갱신에 대한 자세한 내용에 대해 해당 구현을 참조함으로써).

성공하면 0을, 실패하면 -1을 반환합니다.

버전 3.3에 추가.

int **PyState_RemoveModule** (*PyModuleDef* *def)

def에서 만들어진 모듈 객체를 인터프리터 상태에서 제거합니다. 성공하면 0을, 실패하면 -1을 반환합니다.

버전 3.3에 추가.

8.6.3 이터레이터 객체

파이썬은 두 개의 범용 이터레이터 객체를 제공합니다. 첫째, 시퀀스 이터레이터는 *__getitem__()* 메서드를 지원하는 임의의 시퀀스와 작동합니다. 둘째는 콜러블 객체와 종료 신호(sentinel) 값을 사용하고, 시퀀스의 각 항목에 대해 콜러블을 호출하고, 종료 신호 값이 반환될 때 이터레이션을 종료합니다.

PyObject **PySeqIter_Type**

*PySeqIter_New()*와 내장 시퀀스 형에 대한 *iter()* 내장 함수의 단일 인자 형식에 의해 반환된 이터레이터 객체에 대한 형 객체.

int **PySeqIter_Check** (op)

op의 형이 *PySeqIter_Type*이면 참을 돌려줍니다.

*PyObject** **PySeqIter_New** (*PyObject* *seq)

Return value: New reference. 일반 시퀀스 객체 seq와 함께 작동하는 이터레이터를 반환합니다. 시퀀스가 서브스크립션 연산에서 *IndexError*를 일으키면 이터레이션이 끝납니다.

PyObject **PyCallIter_Type**

*PyCallIter_New()*와 *iter()* 내장 함수의 두 인자 형식에 의해 반환된 이터레이터 객체에 대한 형 객체.

int **PyCallIter_Check** (op)

op의 형이 *PyCallIter_Type*이면 참을 돌려줍니다.

*PyObject** **PyCallIter_New** (*PyObject* *callable, *PyObject* *sentinel)

Return value: New reference. 새로운 이터레이터를 돌려줍니다. 첫 번째 매개 변수 *callable*은 매개 변수 없이 호출할 수 있는 모든 파이썬 콜러블 객체일 수 있습니다; 각 호출은 이터레이션의 다음 항목을 반환해야 합니다. *callable*이 *sentinel*와 같은 값을 반환하면 이터레이션이 종료됩니다.

8.6.4 디스크립터 객체

“디스크립터”는 객체의 일부 어트리뷰트를 기술하는 객체입니다. 그것들은 형 객체의 딕셔너리에 있습니다.

PyTypeObject **PyProperty_Type**

내장 디스크립터 형들을 위한 형 객체.

*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)

디스크립터 객체 *descr*가 데이터 어트리뷰트를 기술하고 있으면 참을 반환하고, 메서드를 기술하면 거짓을 돌려줍니다. *descr*는 디스크립터 객체여야 합니다; 오류 검사는 없습니다.

*PyObject** **PyWrapper_New** (*PyObject* *, *PyObject* *)

Return value: New reference.

8.6.5 슬라이스 객체

PyTypeObject **PySlice_Type**

슬라이스 객체의 형 객체. 이것은 파이썬 계층의 *slice*와 같습니다.

int **PySlice_Check** (*PyObject* *ob)

*ob*가 슬라이스 객체면 참을 반환합니다. *ob*는 NULL이 아니어야 합니다.

*PyObject** **PySlice_New** (*PyObject* *start, *PyObject* *stop, *PyObject* *step)

Return value: New reference. 지정된 값으로 새로운 슬라이스 객체를 반환합니다. *start*, *stop* 및 *step* 매개 변수는 같은 이름의 슬라이스 객체 어트리뷰트의 값으로 사용됩니다. 모든 값은 NULL 일 수 있으며, 이 경우 None이 해당 어트리뷰트에 사용됩니다. 새 객체를 할당할 수 없으면 NULL을 반환합니다.

int **PySlice_GetIndices** (*PyObject* *slice, Py_ssize_t length, Py_ssize_t *start, Py_ssize_t *stop, Py_ssize_t *step)

길이가 *length*인 시퀀스를 가정하여, 슬라이스 객체 *slice*에서 *start*, *stop* 및 *step* 인덱스를 가져옵니다. *length* 보다 큰 인덱스를 예외로 처리합니다.

성공하면 0을 반환하고, 예외면 예외 설정 없이 -1을 반환합니다 (인덱스 중 하나가 None이 아니고 정수로 변환되지 않는 한, 이때는 예외를 설정하고 -1을 반환합니다).

이 기능을 사용하고 싶지는 않을 것입니다.

버전 3.2에서 변경: 전에는 *slice* 매개 변수의 매개 변수 형이 *PySliceObject**였습니다.

`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)`

`PySlice_GetIndicesEx()`를 쓸만하게 대체합니다. 길이가 `length`인 시퀀스를 가정하여, 슬라이스 객체 `slice`에서 `start`, `stop` 및 `step` 인덱스를 가져오고, `slicelength`에 슬라이스의 길이를 저장합니다. 범위를 벗어난 인덱스는 일반 슬라이스의 처리와 일관된 방식으로 잘립니다.

성공하면 0을 반환하고, 에러면 예외를 설정하고 -1을 반환합니다.

참고: 이 함수는 크기를 조정할 수 있는 시퀀스에는 안전하지 않은 것으로 간주합니다. 호출은 `PySlice_Unpack()`와 `PySlice_AdjustIndices()`의 조합으로 대체되어야 합니다. 즉

```
if (PySlice_GetIndicesEx(slice, length, &start, &stop, &step, &slicelength) < 0) {
    // return error
}
```

은 다음으로 대체됩니다

```
if (PySlice_Unpack(slice, &start, &stop, &step) < 0) {
    // return error
}
slicelength = PySlice_AdjustIndices(length, &start, &stop, step);
```

버전 3.2에서 변경: 전에는 `slice` 매개 변수의 매개 변수 형이 `PySliceObject*`였습니다.

버전 3.6.1에서 변경: `Py_LIMITED_API`가 설정되어 있지 않거나 `0x03050400`과 `0x03060000` (포함하지 않음) 사이나 `0x03060100` 이상의 값으로 설정되었으면, `PySlice_GetIndicesEx()`는 `PySlice_Unpack()`과 `PySlice_AdjustIndices()`를 사용하는 매크로로 구현됩니다. 인자 `start`, `stop` 및 `step`는 여러 번 평가됩니다.

버전 3.6.1부터 폐지: `Py_LIMITED_API`가 `0x03050400`보다 작거나 `0x03060000`과 `0x03060100` (포함하지 않음) 사이의 값으로 설정되었으면 `PySlice_GetIndicesEx()`는 폐지된 함수입니다.

`int PySlice_Unpack(PyObject *slice, Py_ssize_t *start, Py_ssize_t *stop, Py_ssize_t *step)`

슬라이스 객체의 `start`, `stop` 및 `step` 데이터 멤버를 C 정수로 추출합니다. `PY_SSIZE_T_MAX`보다 큰 값을 `PY_SSIZE_T_MAX`로 조용히 줄이고, `PY_SSIZE_T_MIN`보다 작은 `start`와 `stop` 값을 `PY_SSIZE_T_MIN`로 조용히 높이고, `-PY_SSIZE_T_MAX`보다 작은 `step` 값을 `-PY_SSIZE_T_MAX`로 조용히 높입니다.

에러면 -1을, 성공하면 0을 반환합니다.

버전 3.6.1에 추가.

`Py_ssize_t PySlice_AdjustIndices(Py_ssize_t length, Py_ssize_t *start, Py_ssize_t *stop, Py_ssize_t step)`

지정된 `length` 길이의 시퀀스를 가정하여 `start/stop` 슬라이스 인덱스를 조정합니다. 범위를 벗어난 인덱스는 일반 슬라이스의 처리와 일관된 방식으로 잘립니다.

슬라이스의 길이를 반환합니다. 항상 성공합니다. 파이썬 코드를 호출하지 않습니다.

버전 3.6.1에 추가.

8.6.6 Ellipsis 객체

PyObject *Py_Ellipsis

파이썬 Ellipsis 객체. 이 객체에는 메서드가 없습니다. 참조 횟수와 관련하여 다른 객체와 마찬가지로 처리해야 합니다. *Py_None*와 마찬가지로 싱글톤 객체입니다.

8.6.7 MemoryView 객체

memoryview 객체는 C 수준 버퍼 인터페이스를 다른 객체와 마찬가지로 전달될 수 있는 파이썬 객체로 노출합니다.

PyObject *PyMemoryView_FromObject (*PyObject* *obj)

Return value: New reference. 버퍼 인터페이스를 제공하는 객체에서 memoryview 객체를 만듭니다. *obj*가 쓰기 가능한 버퍼 제공을 지원하면, memoryview 객체는 읽기/쓰기가 되고, 그렇지 않으면 읽기 전용이거나 제공자의 재량에 따라 읽기/쓰기가 될 수 있습니다.

PyObject *PyMemoryView_FromMemory (char *mem, Py_ssize_t size, int flags)

Return value: New reference. *mem*를 하부 버퍼로 사용하여 memoryview 객체를 만듭니다. *flags*는 PyBUF_READ 나 PyBUF_WRITE 중 하나일 수 있습니다.

버전 3.3에 추가.

PyObject *PyMemoryView_FromBuffer (*Py_buffer* *view)

Return value: New reference. 주어진 버퍼 구조체 *view*를 감싸는 memoryview 객체를 만듭니다. 간단한 바이트 버퍼의 경우는, *PyMemoryView_FromMemory()* 가 선호되는 함수입니다.

PyObject *PyMemoryView_GetContiguous (*PyObject* *obj, int buffertype, char order)

Return value: New reference. 버퍼 인터페이스를 정의하는 객체로부터 메모리의 연속 청크('C' 나 'F' or tran order)로 memoryview 객체를 만듭니다. 메모리가 연속적이면 memoryview 객체는 원래 메모리를 가리킵니다. 그렇지 않으면, 복사본이 만들어지고 memoryview는 새 바이트열 객체를 가리킵니다.

int PyMemoryView_Check (*PyObject* *obj)

객체 *obj*가 memoryview 객체면 참을 반환합니다. 현재는 memoryview의 서브 클래스를 만들 수 없습니다.

Py_buffer *PyMemoryView_GET_BUFFER (*PyObject* *mview)

제공자 버퍼의 memoryview의 비공개 복사본의 포인터를 돌려줍니다. *mview*는 반드시 memoryview 인스턴스여야 합니다; 이 매크로는 형을 확인하지 않으므로 직접 검사해야 합니다, 그렇지 않으면 충돌 위험이 있습니다.

Py_buffer *PyMemoryView_GET_BASE (*PyObject* *mview)

memoryview가 기반으로 하는 제공자 객체에 대한 포인터나 memoryview가 *PyMemoryView_FromMemory()* 나 *PyMemoryView_FromBuffer()* 함수 중 하나로 만들어졌으면 NULL을 반환합니다. *mview*는 반드시 memoryview 인스턴스여야 합니다.

8.6.8 약한 참조 객체

파이썬은 약한 참조를 1급 객체로 지원합니다. 약한 참조를 직접 구현하는 두 가지 구체적인 객체 형이 있습니다. 첫 번째는 간단한 참조 객체이며, 두 번째는 가능한 한 원래 객체의 프락시 역할을 합니다.

int PyWeakref_Check (ob)

*ob*가 참조 객체나 프락시 객체면 참을 반환합니다.

int PyWeakref_CheckRef (ob)

*ob*가 참조 객체면 참을 반환합니다.

int PyWeakref_CheckProxy (ob)

*ob*가 프락시 객체면 참을 반환합니다.

*PyObject** **PyWeakref_NewRef** (*PyObject* *ob, *PyObject* *callback)

Return value: New reference. ob 객체에 대한 약한 참조 객체를 반환합니다. 이것은 항상 새로운 참조를 돌려주지만, 새로운 객체를 생성하는 것이 보장되지는 않습니다; 기존 참조 객체가 반환될 수 있습니다. 두 번째 매개 변수인 callback은 ob가 가비지 수집될 때 알림을 받는 콜러블 객체가 될 수 있습니다; 하나의 매개 변수를 받아들여야 하는데, 약한 참조 객체 자체입니다. callback은 None 이나 NULL 일 수도 있습니다. ob가 약하게 참조할 수 있는 객체가 아니거나, callback이 콜러블, None 또는 NULL이 아니면, NULL을 반환하고 TypeError를 발생시킵니다.

*PyObject** **PyWeakref_NewProxy** (*PyObject* *ob, *PyObject* *callback)

Return value: New reference. ob 객체에 대한 약한 참조 프락시 객체를 반환합니다. 이것은 항상 새로운 참조를 돌려주지만, 새로운 객체를 생성하는 것이 보장되지는 않습니다; 기존 프락시 객체가 반환될 수 있습니다. 두 번째 매개 변수인 callback은 ob가 가비지 수집될 때 알림을 받는 콜러블 객체가 될 수 있습니다; 하나의 매개 변수를 받아들여야 하는데, 약한 참조 객체 자체입니다. callback은 None 이나 NULL 일 수도 있습니다. ob가 약하게 참조할 수 있는 객체가 아니거나, callback이 콜러블, None 또는 NULL이 아니면, NULL을 반환하고 TypeError를 발생시킵니다.

*PyObject** **PyWeakref_GetObject** (*PyObject* *ref)

Return value: Borrowed reference. 약한 참조(ref)로부터 참조된 객체를 반환합니다. 참조가 더는 살아있지 않으면, Py_None을 반환합니다.

참고: 이 함수는 참조된 객체에 대한 빌린 참조를 반환합니다. 이는 객체를 계속 사용하는 동안 객체가 파괴될 수 없음을 알고 있을 때를 제외하고, 객체에 대해 항상 `Py_INCREF()` 을 호출해야 함을 뜻합니다.

*PyObject** **PyWeakref_GET_OBJECT** (*PyObject* *ref)

Return value: Borrowed reference. `PyWeakref_GetObject()` 와 유사하지만, 예러 검사를 수행하지 않는 매크로로 구현됩니다.

8.6.9 캡슐

이 객체 사용에 대한 자세한 정보는 using-capsules 를 참조하십시오.

버전 3.1에 추가.

PyCapsule

이 *PyObject*의 서브 형은 불투명한 값을 나타내며, 파이썬 코드를 통해 다른 C 코드로 불투명한 값(void* 포인터로)을 전달해야 하는 C 확장 모듈에 유용합니다. 이것은 한 모듈에서 정의된 C 함수 포인터를 다른 모듈에서 사용할 수 있게 만드는 데 종종 사용되므로, 일반 임포트 메커니즘을 사용하여 동적으로 로드된 모듈에 정의된 C API에 액세스할 수 있습니다.

PyCapsule_Destructor

캡슐에 대한 파괴자(destructor) 콜백 형. 이렇게 정의됩니다:

```
typedef void (*PyCapsule_Destructor) (PyObject *);
```

PyCapsule_Destructor 콜백의 의미는 `PyCapsule_New()` 를 참조하십시오.

int **PyCapsule_CheckExact** (*PyObject* *p)

인자가 *PyCapsule*이면 참을 돌려줍니다.

*PyObject** **PyCapsule_New** (void *pointer, const char *name, *PyCapsule_Destructor* destructor)

Return value: New reference. pointer를 캡슐화하는 *PyCapsule*을 만듭니다. pointer 인자는 NULL이 아닐 수도 있습니다.

실패하면, 예외를 설정하고 NULL을 반환합니다.

name 문자열은 NULL 이나 유효한 C 문자열에 대한 포인터일 수 있습니다. NULL이 아니면, 이 문자열은 캡슐보다 오래 유지되어야 합니다. (destructor 내부에서 해제할 수는 있습니다.)

destructor 인자가 NULL이 아니면, 캡슐이 파괴될 때 캡슐을 인자로 호출됩니다.

이 캡슐을 모듈의 어트리뷰트로 저장하려면, *name*을 `modulename.attributename`로 지정해야 합니다. 이렇게 하면 다른 모듈이 `PyCapsule_Import()`를 사용하여 캡슐을 임포트 할 수 있습니다.

void* PyCapsule_GetPointer (*PyObject* *capsule, const char *name)

캡슐에 저장된 *pointer*를 가져옵니다. 실패하면, 예외를 설정하고 NULL을 반환합니다.

name 매개 변수는 캡슐에 저장된 이름과 정확하게 비교되어야 합니다. 캡슐에 저장된 이름이 NULL이면, 전달된 *name*도 NULL 이어야 합니다. 파이썬은 C 함수 `strcmp()`를 사용하여 캡슐 이름을 비교합니다.

PyCapsule_Destructor **PyCapsule_GetDestructor** (*PyObject* *capsule)

캡슐에 저장된 현재 파괴자를 반환합니다. 실패하면, 예외를 설정하고 NULL을 반환합니다.

캡슐이 NULL 파괴자를 갖는 것은 합법적입니다. 이것은 NULL 반환 코드를 다소 모호하게 만듭니다; 명확히 하려면 `PyCapsule_IsValid()` 나 `PyErr_Occurred()`를 사용하십시오.

void* PyCapsule_GetContext (*PyObject* *capsule)

캡슐에 저장된 현재 컨텍스트를 반환합니다. 실패하면, 예외를 설정하고 NULL을 반환합니다.

캡슐이 NULL 컨텍스트를 갖는 것은 합법적입니다. 이것은 NULL 반환 코드를 다소 모호하게 만듭니다; 명확히 하려면 `PyCapsule_IsValid()` 나 `PyErr_Occurred()`를 사용하십시오.

const char* PyCapsule_GetName (*PyObject* *capsule)

캡슐에 저장된 현재 이름을 반환합니다. 실패하면, 예외를 설정하고 NULL을 반환합니다.

캡슐이 NULL 이름을 갖는 것은 합법적입니다. 이것은 NULL 반환 코드를 다소 모호하게 만듭니다; 명확히 하려면 `PyCapsule_IsValid()` 나 `PyErr_Occurred()`를 사용하십시오.

void* PyCapsule_Import (const char *name, int no_block)

모듈의 캡슐 어트리뷰트에서 C 객체에 대한 포인터를 임포트 합니다. *name* 매개 변수는 `module.attribute` 처럼 어트리뷰트의 전체 이름을 지정해야 합니다. 캡슐에 저장된 *name*은, 이 문자열과 정확히 일치해야 합니다. *no_block*이 참이면, 블록하지 않고 모듈을 임포트 합니다 (`PyImport_ImportModuleNoBlock()`를 사용해서). *no_block*이 거짓이면, 모듈을 평범하게 임포트 합니다 (`PyImport_ImportModule()`을 사용해서).

성공하면 캡슐의 내부 *pointer*를 반환합니다. 실패하면, 예외를 설정하고 NULL를 반환합니다.

int PyCapsule_IsValid (*PyObject* *capsule, const char *name)

capsule 이 유효한 캡슐인지를 판단합니다. 유효한 캡슐은 NULL이 아니며, `PyCapsule_CheckExact()`를 통과하고, NULL이 아닌 포인터가 저장되며, 내부 이름이 *name* 매개 변수와 일치합니다. (캡슐 이름을 비교하는 방법에 대한 정보는 `PyCapsule_GetPointer()`를 참조하십시오.)

즉, `PyCapsule_IsValid()`가 참값을 반환하면, 모든 접근자(`PyCapsule_Get()`으로 시작하는 모든 함수)에 대한 호출이 성공함이 보장됩니다.

객체가 유효하고 전달된 이름과 일치하면 0이 아닌 값을 반환합니다. 그렇지 않으면 0을 반환합니다. 이 함수는 실패하지 않습니다.

int PyCapsule_SetContext (*PyObject* *capsule, void *context)

capsule 내부의 컨텍스트 포인터를 *context*로 설정합니다.

성공하면 0을 반환합니다. 실패하면 0이 아닌 값을 반환하고 예외를 설정합니다.

int PyCapsule_SetDestructor (*PyObject* *capsule, *PyCapsule_Destructor* destructor)

capsule 내부의 파괴자를 *destructor*로 설정합니다.

성공하면 0을 반환합니다. 실패하면 0이 아닌 값을 반환하고 예외를 설정합니다.

int PyCapsule_SetName (*PyObject* *capsule, const char *name)

capsule 내부의 이름을 *name*으로 설정합니다. NULL이 아니면, 이름은 캡슐보다 오래 유지되어야 합니다. 캡슐에 저장된 이전 *name*이 NULL이 아니면, 이를 해제하려고 시도하지 않습니다.

성공하면 0을 반환합니다. 실패하면 0이 아닌 값을 반환하고 예외를 설정합니다.

int PyCapsule_SetPointer (*PyObject* *capsule, void *pointer)

capsule 내부의 void 포인터를 *pointer*로 설정합니다. 포인터는 NULL이 아닐 수 있습니다.

성공하면 0을 반환합니다. 실패하면 0이 아닌 값을 반환하고 예외를 설정합니다.

8.6.10 제너레이터 객체

제너레이터 객체는 파이썬이 제너레이터 이터레이터를 구현하기 위해 사용하는 객체입니다. 일반적으로 *PyGen_New()* 또는 *PyGen_NewWithQualName()*를 명시적으로 호출하는 것이 아니라, 값을 일드(yield)하는 함수를 이터레이트하여 만들어집니다.

PyGenObject

제너레이터 객체에 사용되는 C 구조체.

PyTypeObject **PyGen_Type**

제너레이터 객체에 해당하는 형 객체

int PyGen_Check (*PyObject* *ob)

*ob*가 제너레이터 객체면 참을 돌려줍니다; *ob*는 NULL이 아니어야 합니다.

int PyGen_CheckExact (*PyObject* *ob)

*ob*의 형이 *PyGen_Type*이면 참을 돌려줍니다; *ob*는 NULL이 아니어야 합니다.

*PyObject** **PyGen_New** (*PyFrameObject* *frame)

Return value: New reference. *frame* 객체에 기반한 새 제너레이터 객체를 만들어 반환합니다. 이 함수는 *frame*에 대한 참조를 훔칩니다. 인자는 NULL이 아니어야 합니다.

*PyObject** **PyGen_NewWithQualName** (*PyFrameObject* *frame, *PyObject* *name, *PyObject* *qualname)

Return value: New reference. *frame* 객체에 기반한 새 제너레이터 객체를 만들어 반환하는데, `__name__`과 `__qualname__`를 *name* 및 *qualname*로 설정합니다. 이 함수는 *frame*에 대한 참조를 훔칩니다. *frame* 인자는 NULL이 아니어야 합니다.

8.6.11 코루틴 객체

버전 3.5에 추가.

코루틴 객체는 `async` 키워드로 선언된 함수가 반환하는 것입니다.

PyCoroObject

코루틴 객체에 사용되는 C 구조체.

PyTypeObject **PyCoro_Type**

코루틴 객체에 해당하는 형 객체.

int PyCoro_CheckExact (*PyObject* *ob)

*ob*의 형이 *PyCoro_Type*이면 참을 반환합니다. *ob*는 NULL일 수 없습니다.

*PyObject** **PyCoro_New** (*PyFrameObject* *frame, *PyObject* *name, *PyObject* *qualname)

Return value: New reference. *frame* 객체를 기반으로 새 코루틴 객체를 만들어서 반환합니다. `__name__`과 `__qualname__`은 *name*과 *qualname*로 설정합니다. 이 함수는 *frame*에 대한 참조를 훔칩니다. *frame* 인자는 NULL일 수 없습니다.

8.6.12 컨텍스트 변수 객체

참고: 버전 3.7.1에서 변경: 파이썬 3.7.1에서 모든 컨텍스트 변수 C API의 서명이 *PyContext*, *PyContextVar* 및 *PyContextToken* 대신 *PyObject* 포인터를 사용하도록 변경되었습니다, 예를 들어:

```
// in 3.7.0:
PyContext *PyContext_New(void);

// in 3.7.1+:
PyObject *PyContext_New(void);
```

자세한 내용은 [bpo-34762](#)를 참조하십시오.

버전 3.7에 추가.

이 절에서는 contextvars 모듈을 위한 공용 C API에 대해 자세히 설명합니다.

PyContext

contextvars.Context 객체를 나타내는 데 사용되는 C 구조체.

PyContextVar

contextvars.ContextVar 객체를 나타내는 데 사용되는 C 구조체.

PyContextToken

contextvars.Token 객체를 나타내는 데 사용되는 C 구조체.

PyTypeObject **PyContext_Type**

context 형을 나타내는 형 객체.

PyTypeObject **PyContextVar_Type**

컨텍스트 변수 형을 나타내는 형 객체.

PyTypeObject **PyContextToken_Type**

컨텍스트 변수 토큰 형을 나타내는 형 객체.

형 검사 매크로:

int PyContext_CheckExact (*PyObject* *o)

o가 *PyContext_Type* 형이면 참을 돌려줍니다. o는 NULL이 아니어야 합니다. 이 함수는 항상 성공합니다.

int PyContextVar_CheckExact (*PyObject* *o)

o가 *PyContextVar_Type* 형이면 참을 돌려줍니다. o는 NULL이 아니어야 합니다. 이 함수는 항상 성공합니다.

int PyContextToken_CheckExact (*PyObject* *o)

o가 *PyContextToken_Type* 형이면 참을 돌려줍니다. o는 NULL이 아니어야 합니다. 이 함수는 항상 성공합니다.

컨텍스트 객체 관리 함수:

PyObject ***PyContext_New** (void)

Return value: New reference. 새로운 빈 컨텍스트 객체를 만듭니다. 에러가 발생하면 NULL를 반환합니다.

PyObject ***PyContext_Copy** (*PyObject* *ctx)

Return value: New reference. 전달된 ctx 컨텍스트 객체의 얇은 복사본을 만듭니다. 에러가 발생하면 NULL를 반환합니다.

PyObject *PyContext_CopyCurrent (void)

Return value: New reference. 현재 스레드 컨텍스트의 얇은 복사본을 만듭니다. 에러가 발생하면 NULL을 반환합니다.

int PyContext_Enter (*PyObject* *ctx)

현재 스레드의 현재 컨텍스트로 ctx를 설정합니다. 성공 시 0을 반환하고, 에러 시 -1을 반환합니다.

int PyContext_Exit (*PyObject* *ctx)

ctx 컨텍스트를 비활성화하고 이전 컨텍스트를 현재 스레드의 현재 컨텍스트로 복원합니다. 성공 시 0을 반환하고, 에러 시 -1을 반환합니다.

int PyContext_ClearFreeList ()

컨텍스트 변수 자유 목록을 지웁니다. 해제된 총 항목 수를 반환합니다. 이 함수는 항상 성공합니다.

컨텍스트 변수 함수:

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)

컨텍스트 변수의 값을 가져옵니다. 조회하는 동안 에러가 발생하면 -1을 반환하고, 값이 있는지와 상관없이 에러가 발생하지 않으면 0을 반환합니다.

컨텍스트 변수가 발견되면, value는 그것을 가리키는 포인터가 됩니다. 컨텍스트 변수가 발견되지 않으면, value는 다음을 가리 킵니다:

- default_value, NULL이 아니면;
- var의 기본값, NULL이 아니면;
- 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)

var 컨텍스트 변수의 상태를 token을 반환한 PyContextVar_Set () 호출 전의 상태로 재설정합니다. 이 함수는 성공 시 0을 반환하고, 에러 시 -1을 반환합니다.

8.6.13 DateTime 객체

다양한 날짜와 시간 객체가 datetime 모듈에서 제공됩니다. 이 함수를 사용하기 전에, 헤더 파일 datetime.h가 소스에 포함되어야 하고 (Python.h가 포함하지 않음에 유의하십시오), 일반적으로 모듈 초기화 함수의 일부로 PyDateTime_IMPORT 매크로를 호출해야 합니다. 매크로는 C 구조체에 대한 포인터를 다음 매크로에서 사용되는 static 변수 PyDateTimeAPI에 넣습니다.

UTC 싱글톤에 액세스하기 위한 매크로:

PyObject *PyDateTime_TimeZone_UTC

UTC를 나타내는 시간대 싱글톤을 반환합니다, datetime.timezone.utc와 같은 객체입니다.

버전 3.7에 추가.

형 검사 매크로:

int PyDate_Check (*PyObject* *ob)
ob가 PyDateTime_DateType 형이거나 PyDateTime_DateType의 서브 형이면 참을 돌려줍니다. ob는 NULL이 아니어야 합니다.

int PyDate_CheckExact (*PyObject* *ob)
ob가 PyDateTime_DateType 형이면 참을 돌려줍니다. ob는 NULL이 아니어야 합니다.

int PyDateTime_Check (*PyObject* *ob)
ob가 PyDateTime_DateTimeType 형이거나 PyDateTime_DateTimeType의 서브 형이면 참을 돌려줍니다. ob는 NULL이 아니어야 합니다.

int PyDateTime_CheckExact (*PyObject* *ob)
ob가 PyDateTime_DateTimeType 형이면 참을 돌려줍니다. ob는 NULL이 아니어야 합니다.

int PyTime_Check (*PyObject* *ob)
ob가 PyDateTime_TimeType 형이거나 PyDateTime_TimeType의 서브 형이면 참을 돌려줍니다. ob는 NULL이 아니어야 합니다.

int PyTime_CheckExact (*PyObject* *ob)
ob가 PyDateTime_TimeType 형이면 참을 돌려줍니다. ob는 NULL이 아니어야 합니다.

int PyDelta_Check (*PyObject* *ob)
ob가 PyDateTime_DeltaType 형이거나 PyDateTime_DeltaType의 서브 형이면 참을 돌려줍니다. ob는 NULL이 아니어야 합니다.

int PyDelta_CheckExact (*PyObject* *ob)
ob가 PyDateTime_DeltaType 형이면 참을 돌려줍니다. ob는 NULL이 아니어야 합니다.

int PyTZInfo_Check (*PyObject* *ob)
ob가 PyDateTime_TZInfoType 형이거나 PyDateTime_TZInfoType의 서브 형이면 참을 돌려줍니다. ob는 NULL이 아니어야 합니다.

int PyTZInfo_CheckExact (*PyObject* *ob)
ob가 PyDateTime_TZInfoType 형이면 참을 돌려줍니다. ob는 NULL이 아니어야 합니다.

객체를 만드는 매크로:

*PyObject** **PyDate_FromDate** (int year, int month, int day)
Return value: New reference. 지정된 년, 월, 일의 datetime.date 객체를 반환합니다.

*PyObject** **PyDateTime_FromDateAndTime** (int year, int month, int day, int hour, int minute, int second, int usecond)
Return value: New reference. 지정된 년, 월, 일, 시, 분, 초 및 마이크로초의 datetime.datetime 객체를 반환합니다.

*PyObject** **PyDateTime_FromDateAndTimeAndFold** (int year, int month, int day, int hour, int minute, int second, int usecond, int fold)
Return value: New reference. 지정된 년, 월, 일, 시, 분, 초, 마이크로초 및 fold의 datetime.datetime 객체를 반환합니다.

버전 3.6에 추가.

*PyObject** **PyTime_FromTime** (int hour, int minute, int second, int usecond)
Return value: New reference. 지정된 시, 분, 초 및 마이크로초의 datetime.time 객체를 반환합니다.

*PyObject** **PyTime_FromTimeAndFold** (int hour, int minute, int second, int usecond, int fold)
Return value: New reference. 지정된 시, 분, 초, 마이크로초 및 fold의 datetime.time 객체를 반환합니다.
버전 3.6에 추가.

*PyObject** **PyDelta_FromDSU** (int days, int seconds, int useconds)
Return value: New reference. 지정된 일, 초 및 마이크로초 수를 나타내는 datetime.timedelta 객체를

반환합니다. 결과 마이크로초와 초가 `datetime.timedelta` 객체에 대해 설명된 범위에 있도록 정규화가 수행됩니다.

*PyObject** **PyTimeZone_FromOffset** (*PyDateTime_DeltaType** *offset*)

Return value: New reference. *offset* 인자로 나타내지는 이름이 없는 고정 오프셋의 `datetime.timezone` 객체를 돌려줍니다.

버전 3.7에 추가.

*PyObject** **PyTimeZone_FromOffsetAndName** (*PyDateTime_DeltaType** *offset*, *PyUnicode** *name*)

Return value: New reference. *offset* 인자와 *tzname name*으로 나타내지는 고정 오프셋의 `datetime.timezone` 객체를 돌려줍니다.

버전 3.7에 추가.

날짜 객체에서 필드를 추출하는 매크로. 인자는 서브 클래스(가령 `PyDateTime_DateTime`)를 포함하여 `PyDateTime_Date`의 인스턴스여야 합니다. 인자는 `NULL`이 아니어야 하며, 형은 검사하지 않습니다:

`int PyDateTime_GET_YEAR (PyDateTime_Date *o)`

양의 `int`로, 년을 반환합니다.

`int PyDateTime_GET_MONTH (PyDateTime_Date *o)`

1에서 12까지의 `int`로, 월을 반환합니다.

`int PyDateTime_GET_DAY (PyDateTime_Date *o)`

1에서 31까지의 `int`로, 일을 반환합니다.

날짜 시간 객체에서 필드를 추출하는 매크로. 인자는 서브 클래스를 포함하여 `PyDateTime_DateTime`의 인스턴스여야 합니다. 인자는 `NULL`이 아니어야 하며, 형은 검사하지 않습니다.:

`int PyDateTime_DATE_GET_HOUR (PyDateTime_DateTime *o)`

0부터 23까지의 `int`로, 시를 반환합니다.

`int PyDateTime_DATE_GET_MINUTE (PyDateTime_DateTime *o)`

0부터 59까지의 `int`로, 분을 반환합니다.

`int PyDateTime_DATE_GET_SECOND (PyDateTime_DateTime *o)`

0부터 59까지의 `int`로, 초를 반환합니다.

`int PyDateTime_DATE_GET_MICROSECOND (PyDateTime_DateTime *o)`

0부터 999999까지의 `int`로, 마이크로초를 반환합니다.

`int PyDateTime_DATE_GET_FOLD (PyDateTime_DateTime *o)`

Return the fold, as an `int` from 0 through 1.

버전 3.6에 추가.

시간 객체에서 필드를 추출하는 매크로. 인자는 서브 클래스를 포함하여 `PyDateTime_Time`의 인스턴스여야 합니다. 인자는 `NULL`이 아니어야 하며 형은 검사하지 않습니다:

`int PyDateTime_TIME_GET_HOUR (PyDateTime_Time *o)`

0부터 23까지의 `int`로, 시를 반환합니다.

`int PyDateTime_TIME_GET_MINUTE (PyDateTime_Time *o)`

0부터 59까지의 `int`로, 분을 반환합니다.

`int PyDateTime_TIME_GET_SECOND (PyDateTime_Time *o)`

0부터 59까지의 `int`로, 초를 반환합니다.

`int PyDateTime_TIME_GET_MICROSECOND (PyDateTime_Time *o)`

0부터 999999까지의 `int`로, 마이크로초를 반환합니다.

`int PyDateTime_TIME_GET_FOLD (PyDateTime_Time *o)`

Return the fold, as an `int` from 0 through 1.

버전 3.6에 추가.

시간 델타 객체에서 필드를 추출하는 매크로. 인자는 서브 클래스를 포함하여 `PyDateTime_Delta`의 인스턴스여야 합니다. 인자는 `NULL`이 아니어야 하며 형은 검사하지 않습니다.:

`int PyDateTime_DELTA_GET_DAYS (PyDateTime_Delta *o)`
-999999999에서 999999999까지의 `int`로, 일 수를 반환합니다.

버전 3.3에 추가.

`int PyDateTime_DELTA_GET_SECONDS (PyDateTime_Delta *o)`
0부터 86399까지의 `int`로, 초 수를 반환합니다.

버전 3.3에 추가.

`int PyDateTime_DELTA_GET_MICROSECONDS (PyDateTime_Delta *o)`
0에서 999999까지의 `int`로, 마이크로초 수를 반환합니다.

버전 3.3에 추가.

DB API를 구현하는 모듈의 편의를 위한 매크로:

*PyObject** **PyDateTime_FromTimestamp** (*PyObject* *args)
Return value: New reference. `datetime.datetime.fromtimestamp()`에 전달하는 데 적합한 인자 튜플로 새 `datetime.datetime` 객체를 만들고 반환합니다.

*PyObject** **PyDate_FromTimestamp** (*PyObject* *args)
Return value: New reference. `datetime.date.fromtimestamp()`에 전달하는 데 적합한 인자 튜플로 새 `datetime.date` 객체를 만들고 반환합니다.

초기화, 파이널리제이션 및 스레드

파이썬 초기화 구성도 참조하십시오.

9.1 파이썬 초기화 전

파이썬을 내장한 응용 프로그램에서는, 다른 파이썬/C API 함수를 사용하기 전에 `Py_Initialize()` 함수를 호출해야 합니다; 몇 가지 함수와 전역 구성 변수는 예외입니다.

파이썬이 초기화되기 전에 다음 함수를 안전하게 호출할 수 있습니다:

- 구성 함수:
 - `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()`

- 정보 함수:

- `Py_IsInitialized()`
- `PyMem_GetAllocator()`
- `PyObject_GetArenaAllocator()`
- `Py_GetBuildInfo()`
- `Py_GetCompiler()`
- `Py_GetCopyright()`
- `Py_GetPlatform()`
- `Py_GetVersion()`

- 유틸리티:

- `Py_DecodeLocale()`

- 메모리 할당자:

- `PyMem_RawMalloc()`
 - `PyMem_RawRealloc()`
 - `PyMem_RawCalloc()`
 - `PyMem_RawFree()`

참고: 다음 함수는 `Py_Initialize()` 전에 호출하면 안 됩니다: `Py_EncodeLocale()`, `Py_GetPath()`, `Py_GetPrefix()`, `Py_GetExecPrefix()`, `Py_GetProgramFullPath()`, `Py_GetPythonHome()`, `Py_GetProgramName()` 및 `PyEval_InitThreads()`.

9.2 전역 구성 변수

파이썬에는 다양한 기능과 옵션을 제어하기 위한 전역 구성 변수가 있습니다. 기본적으로, 이러한 플래그는 명령 줄 옵션에 의해 제어됩니다.

옵션에 의해 플래그가 설정되면, 플래그 값은 옵션이 설정된 횟수입니다. 예를 들어, `-b`는 `Py_BytesWarningFlag`를 1로 설정하고 `-bb`는 `Py_BytesWarningFlag`를 2로 설정합니다.

int `Py_BytesWarningFlag`

bytes나 bytearray와 str을, 또는 bytes를 int와 비교할 때 경고를 발행합니다. 2보다 크거나 같으면 에러를 발행합니다.

`-b` 옵션으로 설정합니다.

int `Py_DebugFlag`

구문 분석기 디버깅 출력을 켭니다(전문가 전용, 컴파일 옵션에 의존합니다).

`-d` 옵션과 `PYTHONDEBUG` 환경 변수로 설정됩니다.

int `Py_DontWriteBytecodeFlag`

0이 아닌 값으로 설정하면, 파이썬은 소스 모듈을 임포트 할 때 `.pyc` 파일을 쓰려고 하지 않습니다.

`-B` 옵션과 `PYTHONDONTWRITEBYTECODE` 환경 변수로 설정됩니다.

int `Py_FrozenFlag`

`Py_GetPath()`에서 모듈 검색 경로를 계산할 때 에러 메시지를 표시하지 않습니다.

`_freeze_importlib`와 `frozenmain` 프로그램에서 사용하는 개인 플래그입니다.

int Py_HashRandomizationFlag

PYTHONHASHSEED 환경 변수가 비어 있지 않은 문자열로 설정되면 1로 설정합니다.

플래그가 0이 아니면, PYTHONHASHSEED 환경 변수를 읽어 비밀 해시 시드를 초기화합니다.

int Py_IgnoreEnvironmentFlag

설정될 수 있는 모든 PYTHON* 환경 변수 (예를 들어 PYTHONPATH와 PYTHONHOME)를 무시합니다.

-E와 -I 옵션으로 설정됩니다.

int Py_InspectFlag

스크립트가 첫 번째 인자로 전달되거나 -c 옵션을 사용할 때, sys.stdin가 터미널로 보이지 않더라도 스크립트나 명령을 실행한 후 대화 형 모드로 들어갑니다.

-i 옵션과 PYTHONINSPECT 환경 변수로 설정됩니다.

int Py_InteractiveFlag

-i 옵션으로 설정됩니다.

int Py_IsolatedFlag

격리 모드로 파이썬을 실행합니다. 격리 모드에서 sys.path는 스크립트의 디렉터리도 사용자의 site-packages 디렉터리도 포함하지 않습니다.

-I 옵션으로 설정됩니다.

버전 3.4에 추가.

int Py_LegacyWindowsFSEncodingFlag

플래그가 0이 아니면, 파일 시스템 인코딩에 UTF-8 인코딩 대신 mbcS 인코딩을 사용합니다.

PYTHONLEGACYWINDOWSFSENCODING 환경 변수가 비어 있지 않은 문자열로 설정되면 1로 설정합니다.

자세한 내용은 [PEP 529](#)를 참조하십시오.

가용성: 윈도우.

int Py_LegacyWindowsStdioFlag

플래그가 0이 아니면, sys 표준 스트림을 위해 WindowsConsoleIO 대신 io.FileIO를 사용합니다.

PYTHONLEGACYWINDOWSSSTDIO 환경 변수가 비어 있지 않은 문자열로 설정되면 1로 설정합니다.

자세한 내용은 [PEP 528](#)을 참조하십시오.

가용성: 윈도우.

int Py_NoSiteFlag

모듈 site 임포트와 이에 수반되는 sys.path의 사이트 종속적인 조작을 비활성화합니다. 또한 나중에 site를 명시적으로 임포트 할 때도 이러한 조작을 비활성화합니다 (트리거 하려면 site.main() 을 호출하십시오).

-S 옵션으로 설정됩니다.

int Py_NoUserSiteDirectory

사용자 site-packages 디렉터리를 sys.path에 추가하지 않습니다.

-s와 -I 옵션, 그리고 PYTHONNOUSERSITE 환경 변수로 설정됩니다.

int Py_OptimizeFlag

-O 옵션과 PYTHONOPTIMIZE 환경 변수로 설정됩니다.

int Py_QuietFlag

대화형 모드에서도 저작권과 버전 메시지를 표시하지 않습니다.

-q 옵션으로 설정됩니다.

버전 3.2에 추가.

int Py_UnbufferedStdioFlag

stdout과 stderr 스트림을 버퍼링 해제하도록 강제합니다.

-u 옵션과 PYTHONUNBUFFERED 환경 변수로 설정됩니다.

int Py_VerboseFlag

모듈이 초기화될 때마다, 로드된 위치(파일명이나 내장 모듈)를 표시하는 메시지를 인쇄합니다. 2보다 크거나 같으면, 모듈을 검색할 때 검사되는 각 파일에 대한 메시지를 인쇄합니다. 또한 종료 시 모듈 정리에 대한 정보를 제공합니다.

-v 옵션과 PYTHONVERBOSE 환경 변수로 설정됩니다.

9.3 인터프리터 초기화와 파이널리제이션

void Py_Initialize()

파이썬 인터프리터를 초기화합니다. 파이썬을 내장하는 응용 프로그램에서는, 다른 파이썬/C API 함수를 사용하기 전에 호출해야 합니다; 몇 가지 예외는 [파이썬 초기화 전](#)을 참조하십시오.

이것은 로드된 모듈의 테이블(sys.modules)을 초기화하고, 기반 모듈 builtins, __main__ 및 sys를 만듭니다. 또한, 모듈 검색 경로(sys.path)를 초기화합니다. sys.argv는 설정하지 않습니다; 이를 위해서는 `PySys_SetArgvEx()`를 사용하십시오. (`Py_FinalizeEx()`를 먼저 호출하지 않고) 두 번째로 호출하면 아무런 일도 하지 않습니다. 반환 값이 없습니다; 초기화에 실패하면 치명적인 에러입니다.

참고: 윈도우에서, 콘솔 모드를 O_TEXT에서 O_BINARY로 변경합니다, C 런타임을 사용하는 콘솔의 비파이썬 사용에도 영향을 미칩니다.

void Py_InitializeEx(int initsigs)

이 함수는 `initsigs`가 1이면 `Py_Initialize()`처럼 작동합니다. `initsigs`가 0이면, 시그널 처리기의 초기화 등록을 건너뛰는데, 파이썬이 내장될 때 유용할 수 있습니다.

int Py_IsInitialized()

파이썬 인터프리터가 초기화되었으면 참(0이 아님)을 반환하고, 그렇지 않으면 거짓(0)을 반환합니다. `Py_FinalizeEx()`가 호출된 후, `Py_Initialize()`가 다시 호출될 때까지 거짓을 반환합니다.

int Py_FinalizeEx()

`Py_Initialize()`와 후속 파이썬/C API 함수 사용에 의해 수행된 모든 초기화를 실행 취소하고, `Py_Initialize()`에 대한 마지막 호출 이후 만들어졌지만, 아직 삭제되지 않은 모든 서브 인터프리터(아래 `Py_NewInterpreter()`를 참조하십시오)를 제거합니다. 이상적으로, 이것은 파이썬 인터프리터가 할당한 모든 메모리를 해제합니다. (먼저 `Py_Initialize()`를 다시 호출하지 않고) 두 번째로 호출하면 아무런 일도 하지 않습니다. 일반적으로 반환 값은 0입니다. 파이널리제이션 도중 에러가 발생하면 (버퍼링 된 데이터 플러시) -1이 반환됩니다.

이 함수는 여러 가지 이유로 제공됩니다. 내장 응용 프로그램이 응용 프로그램 자체를 다시 시작하지 않고 파이썬을 다시 시작하고 싶을 수 있습니다. 동적으로 로드할 수 있는 라이브러리(또는 DLL)에서 파이썬 인터프리터를 로드한 응용 프로그램은 DLL을 언로드 하기 전에 파이썬이 할당한 모든 메모리를 해제하고 싶을 수 있습니다. 응용 프로그램에서 메모리 누수를 찾는 동안 개발자는 응용 프로그램을 종료하기 전에 파이썬에서 할당한 모든 메모리를 해제하고 싶을 것입니다.

버그와 주의 사항: 모듈의 모듈과 객체 파괴는 임의의 순서로 수행됩니다; 이로 인해 파괴자(`__del__()` 메서드)가 다른 객체(함수조차)나 모듈에 의존할 때 실패할 수 있습니다. 파이썬에서 로드한 동적으로 로드된 확장 모듈은 언로드 되지 않습니다. 파이썬 인터프리터가 할당한 소량의 메모리는 해제되지 않을 수 있습니다 (누수를 발견하면, 보고해 주십시오). 객체 간의 순환 참조에 묶여있는 메모리는 해제되지 않습니다. 확장 모듈이 할당한 일부 메모리는 해제되지 않을 수 있습니다. 일부 확장은 초기화

루틴이 두 번 이상 호출되면 제대로 작동하지 않을 수 있습니다; 응용 프로그램이 `Py_Initialize()`와 `Py_FinalizeEx()`를 두 번 이상 호출하면 이 문제가 발생할 수 있습니다.

인자 없이 감사 이벤트 `cpython._PySys_ClearAuditHooks`를 발생시킵니다.

버전 3.6에 추가.

void **Py_Finalize()**

이것은 `Py_FinalizeEx()`의 이전 버전과 호환되는 반환 값을 무시하는 버전입니다.

9.4 프로세스 전체 매개 변수

int **Py_SetStandardStreamEncoding**(const char *encoding, const char *errors)

이 함수는 (호출한다면) `Py_Initialize()` 전에 호출해야 합니다. `str.encode()`에서와 같은 의미로, 표준 IO에 사용할 인코딩과 에러 처리를 지정합니다.

PYTHONIOENCODING 값을 재정의(overrides)하고, 환경 변수가 작동하지 않을 때 내장(embedding) 코드가 IO 인코딩을 제어할 수 있도록 합니다.

`encoding` 및/또는 `errors`는 PYTHONIOENCODING 및/또는 기본값(다른 설정에 따라 다릅니다)을 사용하기 위해 NULL일 수 있습니다.

`sys.stderr`은 이 (또는 다른) 설정과 관계없이 항상 “backslashreplace” 에러 처리기를 사용함에 유의하십시오.

`Py_FinalizeEx()`가 호출되면, 이 함수는 `Py_Initialize()`에 대한 후속 호출에 영향을 미치기 위해 다시 호출되어야 합니다.

성공하면 0을 반환하고, 에러 시 (예를 들어 인터프리터가 이미 초기화된 후 호출) 0이 아닌 값을 반환합니다.

버전 3.4에 추가.

void **Py_SetProgramName**(const wchar_t *name)

(호출된다면) 이 함수는 `Py_Initialize()`가 처음으로 호출되기 전에 호출되어야 합니다. 인터프리터에게 프로그램의 `main()` 함수에 대한 `argv[0]` 인자의 값을 알려줍니다 (와이드 문자로 변환됩니다). 이것은 `Py_GetPath()`와 아래의 다른 함수에서 인터프리터 실행 파일과 관련된 파이썬 런타임 라이브러리를 찾는 데 사용됩니다. 기본값은 'python'입니다. 인자는 프로그램을 실행하는 동안 내용이 변경되지 않는 정적 저장소의 0으로 끝나는 와이드 문자열을 가리켜야 합니다. 파이썬 인터프리터의 코드는 이 저장소의 내용을 변경하지 않습니다.

바이트 문자열을 디코딩하여 `wchar_t` 문자열을 얻는데 `Py_DecodeLocale()`을 사용합니다.

wchar_t* **Py_GetProgramName**()

`Py_SetProgramName()`으로 설정된 프로그램 이름이나 기본값을 반환합니다. 반환된 문자열은 정적 저장소를 가리킵니다; 호출자는 값을 수정해서는 안 됩니다.

wchar_t* **Py_GetPrefix**()

설치된 플랫폼 독립적 파일에 대한 `prefix`를 반환합니다. 이것은 `Py_SetProgramName()`으로 설정된 프로그램 이름과 일부 환경 변수의 여러 복잡한 규칙을 통해 파생됩니다; 예를 들어, 프로그램 이름이 '/usr/local/bin/python'이면, `prefix`는 '/usr/local'입니다. 반환된 문자열은 정적 저장소를 가리킵니다; 호출자는 값을 수정해서는 안 됩니다. 이는 최상위 수준 Makefile의 `prefix` 변수와 빌드시 `configure` 스크립트의 `--prefix` 인자에 해당합니다. 이 값은 파이썬 코드에서 `sys.prefix`로 사용할 수 있습니다. 유닉스에서만 유용합니다. 다음 함수도 참조하십시오.

wchar_t* **Py_GetExecPrefix**()

설치된 플랫폼-종속적 파일에 대한 `exec-prefix`를 반환합니다. 이것은 `Py_SetProgramName()`으로 설정된 프로그램 이름과 일부 환경 변수의 여러 복잡한 규칙을 통해 파생됩니다; 예를 들어 프로그램 이름이 '/usr/local/bin/python'이면, `exec-prefix`는 '/usr/local'입니다. 반환된 문자열은 정적 저장

소를 가리 킵니다; 호출자는 값을 수정해서는 안 됩니다. 이는 최상위 수준 Makefile의 `exec_prefix` 변수와 빌드 시 `configure` 스크립트의 `--exec-prefix` 인자에 해당합니다. 이 값은 파이썬 코드에서 `sys.exec_prefix`로 사용할 수 있습니다. 유닉스에서만 유용합니다.

배경: `exec-prefix`는 플랫폼 종속적 파일(가령 실행 파일과 공유 라이브러리)이 다른 디렉터리 트리에 설치될 때 `prefix`와 다릅니다. 일반 설치에서, 플랫폼 종속적 파일은 `/usr/local/plat` 서브 트리에 설치되고 플랫폼 독립적 파일은 `/usr/local`에 설치될 수 있습니다.

일반적으로 말해서, 플랫폼은 하드웨어와 소프트웨어 제품군의 조합입니다, 예를 들어 Solaris 2.x 운영 체제를 실행하는 Sparc 기계들은 같은 플랫폼으로 간주하지만, Solaris 2.x를 실행하는 Intel 기계는 다른 플랫폼이며, 리눅스를 실행하는 Intel 기계는 또 다른 플랫폼입니다. 같은 운영 체제의 서로 다른 주 개정판도 일반적으로 다른 플랫폼을 형성합니다. 비 유닉스 운영 체제는 다른 이야기입니다; 이러한 시스템의 설치 전략이 너무 다르기 때문에 `prefix`와 `exec-prefix`는 의미가 없으며, 빈 문자열로 설정됩니다. 컴파일된 파이썬 바이트 코드 파일은 플랫폼 독립적임에 유의하십시오 (그러나 이들을 컴파일하는데 사용된 파이썬 버전에는 종속적입니다!).

시스템 관리자는 `/usr/local/plat`을 각 플랫폼에 대해 다른 파일 시스템으로 사용하면서 플랫폼 간에 `/usr/local`을 공유하도록 `mount`나 `automount` 프로그램을 구성하는 방법을 알 것입니다.

`wchar_t*` **Py_GetProgramFullPath()**

파이썬 실행 파일의 전체 프로그램 이름을 반환합니다; 이것은 프로그램 이름(위의 `Py_SetProgramName()` 으로 설정됩니다)에서 기본 모듈 검색 경로를 파생하는 부작용으로 계산됩니다. 반환된 문자열은 정적 저장소를 가리 킵니다; 호출자는 값을 수정해서는 안 됩니다. 이 값은 파이썬 코드에서 `sys.executable`로 사용할 수 있습니다.

`wchar_t*` **Py_GetPath()**

기본 모듈 검색 경로를 반환합니다; 이것은 프로그램 이름(위의 `Py_SetProgramName()` 으로 설정됩니다)과 일부 환경 변수에서 계산됩니다. 반환된 문자열은 플랫폼 종속적 구분자로 분할된 일련의 디렉터리 이름으로 구성됩니다. 구분자는 유닉스와 Mac OS X에서는 ':', 윈도우에서는 ';'입니다. 반환된 문자열은 정적 저장소를 가리 킵니다; 호출자는 값을 수정해서는 안 됩니다. 리스트 `sys.path`는 인터프리터 시작 시 이 값으로 초기화됩니다; 모듈을 로드하기 위한 검색 경로를 변경하기 위해 나중에 수정할 수 있습니다 (그리고 보통 그렇게 합니다).

`void` **Py_SetPath**(const `wchar_t*`)

기본 모듈 검색 경로를 설정합니다. 이 함수가 `Py_Initialize()` 이전에 호출되면, `Py_GetPath()`는 기본 검색 경로를 계산하지 않고 대신 제공된 경로를 사용합니다. 이는 모든 모듈의 위치를 완전히 알고 있는 응용 프로그램에 파이썬이 내장된 경우 유용합니다. 경로 구성 요소는 플랫폼 종속적 구분자 문자(유닉스에서는 ':', 윈도우에서는 ';')로 구분해야 합니다.

또한 `sys.executable`이 프로그램 전체 경로(`Py_GetProgramFullPath()` 를 참조하십시오)로 설정되고 `sys.prefix`와 `sys.exec_prefix`가 비어있도록 합니다. `Py_Initialize()`를 호출한 후 필요할 때 이를 수정하는 것은 호출자에게 달려 있습니다.

바이트 문자열을 디코딩하여 `wchar_t*` 문자열을 얻는데 `Py_DecodeLocale()`을 사용합니다.

경로 인자는 내부적으로 복사되므로, 호출이 완료된 후 호출자가 할당 해제할 수 있습니다.

버전 3.8에서 변경: 이제 프로그램 이름 대신 프로그램 전체 경로가 `sys.executable`에 사용됩니다.

`const char*` **Py_GetVersion()**

이 파이썬 인터프리터의 버전을 반환합니다. 이것은 다음과 같은 문자열입니다

```
"3.0a5+ (py3k:63103M, May 12 2008, 00:53:55) \n[GCC 4.2.3]"
```

첫 번째 단어(첫 번째 스페이스 문자까지)는 현재 파이썬 버전입니다; 처음 세 문자는 마침표로 구분된 주 버전과 부 버전입니다. 반환된 문자열은 정적 저장소를 가리 킵니다; 호출자는 값을 수정해서는 안 됩니다. 이 값은 파이썬 코드에서 `sys.version`으로 사용할 수 있습니다.

`const char*` **Py_GetPlatform()**

현재 플랫폼의 플랫폼 식별자를 반환합니다. 유닉스에서, 이것은 운영 체제의 “공식적인” 이름으로

구성되며, 소문자로 변환되고, 그 뒤에 주 개정 번호가 붙습니다; 예를 들어, SunOS 5.x라고도 하는 Solaris 2.x의 경우, 값은 'sunos5'입니다. Mac OS X에서는, 'darwin'입니다. 윈도우에서는, 'win'입니다. 반환된 문자열은 정적 저장소를 가리킵니다; 호출자는 값을 수정해서는 안 됩니다. 이 값은 파이썬 코드에서 `sys.platform`으로 사용할 수 있습니다.

`const char* Py_GetCopyright()`

현재 파이썬 버전에 대한 공식 저작권 문자열을 반환합니다, 예를 들어

'Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam'

반환된 문자열은 정적 저장소를 가리킵니다; 호출자는 값을 수정해서는 안 됩니다. 이 값은 파이썬 코드에서 `sys.copyright`로 사용할 수 있습니다.

`const char* Py_GetCompiler()`

현재 파이썬 버전을 빌드하는 데 사용된 컴파일러 표시를 대괄호 감싸서 반환합니다, 예를 들면:

```
"[GCC 2.7.2.2]"
```

반환된 문자열은 정적 저장소를 가리킵니다; 호출자는 값을 수정해서는 안 됩니다. 이 값은 파이썬 코드에서 변수 `sys.version`의 일부로 제공됩니다.

`const char* Py_BuildInfo()`

현재 파이썬 인터프리터 인스턴스의 시퀀스 번호와 빌드 날짜 및 시간에 대한 정보를 반환합니다, 예를 들어

```
"#67, Aug 1 1997, 22:34:28"
```

반환된 문자열은 정적 저장소를 가리킵니다; 호출자는 값을 수정해서는 안 됩니다. 이 값은 파이썬 코드에서 변수 `sys.version`의 일부로 제공됩니다.

`void PySys_SetArgvEx(int argc, wchar_t**argv, int updatepath)`

`argc` 및 `argv`에 기반해서 `sys.argv`를 설정합니다. 이 매개 변수는 프로그램의 `main()` 함수에 전달된 것과 유사하지만, 첫 번째 항목이 파이썬 인터프리터를 호스팅하는 실행 파일이 아니라 실행될 스크립트 파일을 참조해야 한다는 차이점이 있습니다. 실행할 스크립트가 없으면, `argv`의 첫 번째 항목은 빈 문자열일 수 있습니다. 이 함수가 `sys.argv` 초기화에 실패하면, `Py_FatalError()`를 사용하여 치명적인 조건을 표시합니다.

`updatepath`가 0이면, 여기까지가 이 함수가 하는 모든 일입니다. `updatepath`가 0이 아니면, 함수는 다음 알고리즘에 따라 `sys.path`도 수정합니다:

- 기존 스크립트의 이름이 `argv[0]`으로 전달되면, 스크립트가 있는 디렉터리의 절대 경로가 `sys.path` 앞에 추가됩니다.
- 그렇지 않으면 (즉, `argc`가 0이거나 `argv[0]`이 기존 파일 이름을 가리키지 않으면), `sys.path` 앞에 빈 문자열이 추가됩니다, 이는 현재 작업 디렉터리(".")를 앞에 추가하는 것과 같습니다.

바이트 문자열을 디코딩하여 `wchar_t*` 문자열을 얻는데 `Py_DecodeLocale()`을 사용합니다.

참고: 단일 스크립트 실행 이외의 목적으로 파이썬 인터프리터를 내장하는 응용 프로그램은 0을 `updatepath`로 전달하고, 원하는 대로 `sys.path`를 스스로 갱신하는 것이 좋습니다. CVE-2008-5983을 참조하십시오.

3.1.3 이전 버전에서는, `PySys_SetArgv()`를 호출한 후 첫 번째 `sys.path` 요소를 수동으로 제거하여 같은 효과를 얻을 수 있습니다, 예를 들어 다음을 사용하여:

```
PyRun_SimpleString("import sys; sys.path.pop(0)\n");
```

버전 3.1.3에 추가.

`void PySys_SetArgv (int argc, wchar_t **argv)`

이 함수는 `python` 인터프리터가 `-I`로 시작되지 않는 한 `updatepath`가 1로 설정된 `PySys_SetArgvEx()`처럼 작동합니다.

바이트 문자열을 디코딩하여 `wchar_t*` 문자열을 얻는데 `Py_DecodeLocale()`을 사용합니다.

버전 3.4에서 변경: `updatepath` 값은 `-I`에 따라 다릅니다.

`void Py_SetPythonHome (const wchar_t *home)`

기본 “홈” 디렉터리, 즉 표준 파이썬 라이브러리의 위치를 설정합니다. 인자 문자열의 의미는 `PYTHONHOME`을 참조하십시오.

인자는 프로그램을 실행하는 동안 내용이 변경되지 않는 정적 저장소에 있는 0으로 끝나는 문자열을 가리켜야 합니다. 파이썬 인터프리터의 코드는 이 저장소의 내용을 변경하지 않습니다.

바이트 문자열을 디코딩하여 `wchar_t*` 문자열을 얻는데 `Py_DecodeLocale()`을 사용합니다.

`w_char* Py_GetPythonHome ()`

기본 “홈”, 즉 `Py_SetPythonHome()`에 대한 이전 호출에서 설정한 값이나 설정되었다면 `PYTHONHOME` 환경 변수의 값을 반환합니다.

9.5 스레드 상태와 전역 인터프리터 록

파이썬 인터프리터는 완전히 스레드 안전하지 않습니다. 다중 스레드 파이썬 프로그램을 지원하기 위해, 파이썬 객체에 안전하게 액세스하기 전에 현재 스레드가 보유해야 하는 **전역 인터프리터 록** 혹은 **GIL**이라고 하는 전역 록이 있습니다. 록 없이는, 가장 간단한 연산조차도 다중 스레드 프로그램에서 문제를 일으킬 수 있습니다: 예를 들어, 두 스레드가 동시에 같은 객체의 참조 횟수를 증가시키면, 참조 횟수가 두 번이 아닌 한 번만 증가할 수 있습니다.

따라서, **GIL**을 획득한 스레드만 파이썬 객체에서 작동하거나 파이썬/C API 함수를 호출할 수 있다는 규칙이 있습니다. 동시 실행을 모방하기 위해 인터프리터는 정기적으로 스레드 전환을 시도합니다 (`sys.setswitchinterval()`을 참조하십시오). 록은 파일 읽기나 쓰기와 같은 잠재적인 블로킹 I/O 연산에 대해서도 해제되므로, 그동안 다른 파이썬 스레드가 실행될 수 있습니다.

파이썬 인터프리터는 `PyThreadState`라는 데이터 구조체 내에 스레드 별 부기(bookkeeping) 정보를 보관합니다. 현재 `PyThreadState`를 가리키는 하나의 전역 변수도 있습니다: `PyThreadState_Get()`을 사용하여 얻을 수 있습니다.

9.5.1 확장 코드에서 GIL 해제하기

GIL을 조작하는 대부분의 확장 코드는 다음과 같은 간단한 구조로 되어 있습니다:

```
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.
```

이것은 매우 일반적이어서 이를 단순화하기 위해 한 쌍의 매크로가 존재합니다:

```
Py_BEGIN_ALLOW_THREADS
... Do some blocking I/O operation ...
Py_END_ALLOW_THREADS
```

`Py_BEGIN_ALLOW_THREADS` 매크로는 새 블록을 열고 숨겨진 지역 변수를 선언합니다; `Py_END_ALLOW_THREADS` 매크로는 블록을 닫습니다.

위의 블록은 다음 코드로 확장됩니다:

```
PyThreadState *_save;

_save = PyEval_SaveThread();
... Do some blocking I/O operation ...
PyEval_RestoreThread(_save);
```

이 함수들의 작동 방식은 다음과 같습니다: 전역 인터프리터 록이 현재 스레드 상태에 대한 포인터를 보호하는데 사용됩니다. 록을 해제하고 스레드 상태를 저장할 때, 록이 해제되기 전에 현재 스레드 상태 포인터를 가져와야 합니다(다른 스레드가 즉시 록을 획득하고 전역 변수에 자신의 스레드 상태를 저장할 수 있기 때문입니다). 반대로, 록을 획득하고 스레드 상태를 복원할 때, 스레드 상태 포인터를 저장하기 전에 록을 획득해야 합니다.

참고: 시스템 I/O 함수 호출은 GIL을 릴리스하는 가장 일반적인 사용 사례이지만, 메모리 버퍼를 통해 작동하는 압축이나 암호화 함수와 같이, 파이썬 객체에 액세스할 필요가 없는 장기 실행 계산을 호출하기 전에도 유용할 수 있습니다. 예를 들어, 표준 `zlib`와 `hashlib` 모듈은 데이터를 압축하거나 해싱할 때 GIL을 해제합니다.

9.5.2 파이썬이 만들지 않은 스레드

전용 파이썬 API(가령 `threading` 모듈)를 사용하여 스레드를 만들면, 스레드 상태가 자동으로 연결되므로 위에 표시된 코드가 올바릅니다. 그러나, 스레드가 C에서 만들어질 때 (예를 들어 자체 스레드 관리 기능이 있는 제삼자 라이브러리에 의해), GIL을 보유하지 않고, 그들을 위한 스레드 상태 구조도 없습니다.

이러한 스레드에서 파이썬 코드를 호출해야 하면 (종종 앞서 언급한 제삼자 라이브러리에서 제공하는 콜백 API의 일부가 됩니다), 먼저 스레드 상태 자료 구조를 만들어서 인터프리터에 이러한 스레드를 등록한 다음, GIL을 획득하고, 마지막으로 파이썬/C API 사용을 시작하기 전에 스레드 상태 포인터를 저장합니다. 완료되면, 스레드 상태 포인터를 재설정하고, GIL을 해제한 다음, 마지막으로 스레드 상태 자료 구조를 해제해야 합니다.

`PyGILState_Ensure()`와 `PyGILState_Release()` 함수는 위의 모든 작업을 자동으로 수행합니다. C 스레드에서 파이썬을 호출하는 일반적인 관용구는 다음과 같습니다:

```
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);
```

`PyGILState_*`() 함수는 (`Py_Initialize()`에 의해 자동으로 만들어진) 전역 인터프리터 하나만 있다고 가정함에 유의하십시오. 파이썬은 추가 인터프리터를 만드는 것을 지원하지만 (`Py_NewInterpreter()`를 사용해서), 다중 인터프리터와 `PyGILState_*`() API를 혼합하는 것은 지원되지 않습니다.

9.5.3 fork()에 대한 주의 사항

스레드에 대해 주목해야 할 또 다른 중요한 점은 C `fork()` 호출 시 스레드의 동작입니다. `fork()`를 사용하는 대부분의 시스템에서는, 프로세스가 포크한 후에 포크를 발행한 스레드만 존재합니다. 이는 록을 처리해야 하는 방법과 CPython 런타임에 저장된 모든 상태 모두에 구체적인 영향을 미칩니다.

“현재” 스레드만 남아 있다는 사실은 다른 스레드가 보유한 록이 해제되지 않음을 의미합니다. 파이썬은 포크 전에 내부적으로 사용하는 록을 획득하고 나중에 해제하여 `os.fork()`에 대해 이 문제를 해결합니다. 또한, 자식의 모든 lock-objects를 재설정합니다. 파이썬을 확장하거나 내장할 때, 포크 이전에 획득하거나 이후에 재설정해야 하는 추가(비 파이썬) 록을 파이썬에 알릴 방법이 없습니다. `pthread_atfork()`와 같은 OS 기능을 사용하여 같은 작업을 수행해야 합니다. 또한, 파이썬을 확장하거나 내장할 때, `os.fork()`를 통하지 않고 직접 `fork()`를 호출하면 (그리고 파이썬으로 반환하거나 파이썬을 호출하면), 포크 이후에 사라지는 스레드가 보유하는 파이썬의 내부 록 중 하나에 의해 교착 상태가 발생할 수 있습니다. `PyOS_AfterFork_Child()`는 필요한 록을 재설정하려고 하지만, 항상 할 수 있는 것은 아닙니다.

다른 모든 스레드가 사라진다는 사실은 또한 CPython의 런타임 상태가 `os.fork()`와 마찬가지로 적절하게 정리되어야 함을 의미합니다. 이것은 현재 인터프리터와 다른 모든 `PyInterpreterState` 객체에 속하는 다른 모든 `PyThreadState` 객체를 파이널리제이션 하는 것을 의미합니다. 이것과 “메인” 인터프리터의 특수한 특성으로 인해, `fork()`는 CPython 전역 런타임이 원래 초기화된 인터프리터의 “메인” 스레드에서만 호출되어야 합니다. 유일한 예외는 `exec()`가 그 후에 즉시 호출되는 경우입니다.

9.5.4 고수준 API

다음은 C 확장 코드를 작성하거나 파이썬 인터프리터를 내장할 때 가장 일반적으로 사용되는 형과 함수입니다:

PyInterpreterState

이 자료 구조는 여러 협력 스레드가 공유하는 상태를 나타냅니다. 같은 인터프리터에 속하는 스레드는 모듈 관리와 몇 가지 다른 내부 항목을 공유합니다. 이 구조체에는 공개 멤버가 없습니다.

다른 인터프리터에 속한 스레드는 사용 가능한 메모리, 열린 파일 기술자 등과 같은 프로세스 상태를 제외하고는, 처음에는 아무것도 공유하지 않습니다. 전역 인터프리터 록은 어떤 인터프리터에 속해 있는지에 관계없이 모든 스레드에서 공유됩니다.

PyThreadState

이 자료 구조는 단일 스레드의 상태를 나타냅니다. 유일한 공용 데이터 멤버는 이 스레드의 인터프리터 상태를 가리키는 `interp(PyInterpreterState *)`입니다.

void PyEval_InitThreads()

Initialize and acquire the global interpreter lock. It should be called in the main thread before creating a second thread or engaging in any other thread operations such as `PyEval_ReleaseThread(tstate)`. It is not needed before calling `PyEval_SaveThread()` or `PyEval_RestoreThread()`.

This is a no-op when called for a second time.

버전 3.7에서 변경: 이 함수는 이제 `Py_Initialize()`에 의해 호출되어서, 여러분은 더는 직접 호출할 필요가 없습니다.

버전 3.2에서 변경: 이 함수는 더는 `Py_Initialize()` 전에 호출할 수 없습니다.

int PyEval_ThreadsInitialized()

`PyEval_InitThreads()`가 호출되었으면, 0이 아닌 값을 반환합니다. 이 함수는 GIL을 보유하지 않고 호출할 수 있어서, 단일 스레드를 실행할 때 록 API 호출을 회피하는 데 사용할 수 있습니다.

버전 3.7에서 변경: `GIL`은 이제 `Py_Initialize()`에 의해 초기화됩니다.

PyThreadState* PyEval_SaveThread()

(만들었다면) 전역 인터프리터 록을 해제하고 스레드 상태를 NULL로 재설정하고, 이전 스레드 상태 (NULL이 아닙니다)를 반환합니다. 록이 만들어졌다면, 현재 스레드가 록을 획득했어야 합니다.

void **PyEval_RestoreThread** (*PyThreadState* *tstate)

(만들었다면) 전역 인터프리터 록을 획득하고 스레드 상태를 NULL이 아니어야 하는 *tstate*로 설정합니다. 록이 만들어졌다면, 현재 스레드가 이를 획득하지 않았어야 합니다, 그렇지 않으면 교착 상태가 발생합니다.

참고: 런타임이 파이널리제이션 될 때 스레드에서 이 함수를 호출하면, 스레드가 파이썬에 의해 만들어지지 않았더라도 스레드가 종료됩니다. 원치 않는 종료를 방지하려면 `_Py_IsFinalizing()` 이나 `sys.is_finalizing()` 을 사용하여 이 함수를 호출하기 전에 인터프리터가 파이널리제이션 되고 있는지 확인할 수 있습니다.

*PyThreadState** **PyThreadState_Get** ()

현재 스레드 상태를 반환합니다. 전역 인터프리터 록을 보유해야 합니다. 현재 스레드 상태가 NULL이면 치명적인 에러가 발생합니다 (그래서 호출자가 NULL을 확인할 필요가 없습니다).

*PyThreadState** **PyThreadState_Swap** (*PyThreadState* *tstate)

현재 스레드 상태를 인자 *tstate*(NULL일 수 있습니다)가 제공하는 스레드 상태와 스와프합니다. 전역 인터프리터 록을 보유해야 하며 해제되지 않습니다.

다음 함수는 스레드 로컬 저장소를 사용하며, 서브 인터프리터와 호환되지 않습니다:

PyGILState_STATE **PyGILState_Ensure** ()

현재 스레드가 파이썬의 현재 상태나 전역 인터프리터 록과 관계없이 파이썬 C API를 호출할 준비가 되었는지 확인합니다. 이것은 각 호출이 `PyGILState_Release()`에 대한 호출과 쌍을 이루는 한 스레드에서 원하는 만큼 여러 번 호출될 수 있습니다. 일반적으로, 스레드 상태가 `Release()` 전에 이전 상태로 복원되는 한 `PyGILState_Ensure()`와 `PyGILState_Release()` 호출 간에 다른 스레드 관련 API를 사용할 수 있습니다. 예를 들어, `Py_BEGIN_ALLOW_THREADS`와 `Py_END_ALLOW_THREADS` 매크로의 정상적인 사용은 허용됩니다.

반환 값은 `PyGILState_Ensure()`가 호출되었을 때의 스레드 상태에 대한 불투명한 “핸들”이며, 파이썬이 같은 상태에 있도록 하려면 `PyGILState_Release()`로 전달되어야 합니다. 재귀 호출이 허용되더라도, 이 핸들들은 공유할 수 없습니다 - `PyGILState_Ensure()`에 대한 각 고유 호출은 자신의 `PyGILState_Release()`에 대한 호출을 위해 핸들을 저장해야 합니다.

함수가 반환할 때, 현재 스레드는 GIL을 보유하고 임의의 파이썬 코드를 호출할 수 있습니다. 실패는 치명적인 에러입니다.

참고: 런타임이 파이널리제이션 될 때 스레드에서 이 함수를 호출하면, 스레드가 파이썬에 의해 만들어지지 않았더라도 스레드가 종료됩니다. 원치 않는 종료를 방지하려면 `_Py_IsFinalizing()` 이나 `sys.is_finalizing()` 을 사용하여 이 함수를 호출하기 전에 인터프리터가 파이널리제이션 되고 있는지 확인할 수 있습니다.

void **PyGILState_Release** (*PyGILState_STATE*)

이전에 획득 한 모든 자원을 해제합니다. 이 호출 후에, 파이썬의 상태는 해당 `PyGILState_Ensure()` 호출 이전과 같습니다 (그러나 일반적으로 이 상태는 호출자에게 알려지지 않아서, GILState API를 사용합니다).

`PyGILState_Ensure()`에 대한 모든 호출은 같은 스레드에서 `PyGILState_Release()`에 대한 호출과 쌍을 이뤄야 합니다.

*PyThreadState** **PyGILState_GetThisThreadState** ()

이 스레드의 현재 스레드 상태를 가져옵니다. 현재 스레드에서 GILState API가 사용되지 않았으면 NULL을 반환할 수 있습니다. 메인 스레드에서 자동 스레드 상태 호출 (auto-thread-state call)이 수행되지 않은 경우에도, 메인 스레드에는 항상 이러한 스레드 상태가 있음에 유의하십시오. 이것은 주로 도우미/진단 함수입니다.

int PyGILState_Check()

현재 스레드가 GIL을 보유하고 있으면 1을 반환하고 그렇지 않으면 0을 반환합니다. 이 함수는 아무 때나 모든 스레드에서 호출할 수 있습니다. 파이썬 스레드 상태가 초기화되었고 현재 GIL을 보유하고 있을 때만 1을 반환합니다. 이것은 주로 도우미/진단 함수입니다. 예를 들어 콜백 컨텍스트나 메모리 할당 함수에서 유용할 수 있는데, GIL이 잠겨 있다는 것을 알면 호출자가 민감한 작업을 수행하거나 그렇지 않으면 다르게 동작하도록 할 수 있습니다.

버전 3.4에 추가.

다음 매크로는 일반적으로 후행 세미콜론 없이 사용됩니다; 파이썬 소스 배포에서 사용 예를 찾으십시오.

Py_BEGIN_ALLOW_THREADS

이 매크로는 { PyThreadState *_save; _save = PyEval_SaveThread(); 로 확장됩니다. 여는 중괄호가 포함되어 있음에 유의하십시오; 뒤따르는 [Py_END_ALLOW_THREADS](#) 매크로와 일치해야 합니다. 이 매크로에 대한 자세한 내용은 위를 참조하십시오.

Py_END_ALLOW_THREADS

이 매크로는 PyEval_RestoreThread(_save); }로 확장됩니다. 닫는 중괄호가 포함되어 있음에 유의하십시오; 이전 [Py_BEGIN_ALLOW_THREADS](#) 매크로와 일치해야 합니다. 이 매크로에 대한 자세한 내용은 위를 참조하십시오.

Py_BLOCK_THREADS

이 매크로는 PyEval_RestoreThread(_save); 로 확장됩니다: 닫는 중괄호가 없는 [Py_END_ALLOW_THREADS](#)와 동등합니다.

Py_UNBLOCK_THREADS

이 매크로는 _save = PyEval_SaveThread(); 로 확장됩니다: 여는 중괄호와 변수 선언이 없는 [Py_BEGIN_ALLOW_THREADS](#)와 동등합니다.

9.5.5 저수준 API

다음 함수는 모두 [Py_Initialize\(\)](#) 이후에 호출되어야 합니다.

버전 3.7에서 변경: [Py_Initialize\(\)](#)는 이제 [GIL](#)을 초기화합니다.

PyInterpreterState* PyInterpreterState_New()

새 인터프리터 상태 객체를 만듭니다. 전역 인터프리터 록을 보유할 필요는 없지만, 이 함수에 대한 호출을 직렬화해야 하면 보유할 수 있습니다.

인자 없이 감사 이벤트 cpython.PyInterpreterState_New를 발생시킵니다.

void PyInterpreterState_Clear(PyInterpreterState *interp)

인터프리터 상태 객체의 모든 정보를 재설정합니다. 전역 인터프리터 록을 보유해야 합니다.

인자 없이 감사 이벤트 cpython.PyInterpreterState_Clear를 발생시킵니다.

void PyInterpreterState_Delete(PyInterpreterState *interp)

인터프리터 상태 객체를 파괴합니다. 전역 인터프리터 록은 보유할 필요 없습니다. 인터프리터 상태는 [PyInterpreterState_Clear\(\)](#)에 대한 이전 호출로 재설정되었어야 합니다.

PyThreadState* PyThreadState_New(PyInterpreterState *interp)

주어진 인터프리터 객체에 속하는 새 스레드 상태 객체를 만듭니다. 전역 인터프리터 록을 보유할 필요는 없지만, 이 함수에 대한 호출을 직렬화해야 하면 보유할 수 있습니다.

void PyThreadState_Clear(PyThreadState *tstate)

스레드 상태 객체의 모든 정보를 재설정합니다. 전역 인터프리터 록을 보유해야 합니다.

void PyThreadState_Delete(PyThreadState *tstate)

스레드 상태 객체를 파괴합니다. 전역 인터프리터 록은 보유할 필요 없습니다. 스레드 상태는 [PyThreadState_Clear\(\)](#)에 대한 이전 호출로 재설정되었어야 합니다.

Py_INT64_T PyInterpreterState_GetID (*PyInterpreterState *interp*)

인터프리터의 고유 ID를 반환합니다. 그렇게 하는데 에러가 발생하면 -1이 반환되고 에러가 설정됩니다.

버전 3.7에 추가.

PyObject* PyInterpreterState_GetDict (*PyInterpreterState *interp*)

인터프리터별 데이터가 저장될 수 있는 딕셔너리를 반환합니다. 이 함수가 NULL을 반환하면 예외는 발생하지 않았고 호출자는 인터프리터별 딕셔너리를 사용할 수 없다고 가정해야 합니다.

이것은 확장이 인터프리터별 상태 정보를 저장하는 데 사용해야 하는 *PyModule_GetState()*를 대체하는 것이 아닙니다.

버전 3.8에 추가.

PyObject* PyThreadState_GetDict ()

Return value: Borrowed reference. 확장이 스레드별 상태 정보를 저장할 수 있는 딕셔너리를 반환합니다. 각 확장은 딕셔너리에 상태를 저장하는 데 사용할 고유 키를 사용해야 합니다. 현재 스레드 상태를 사용할 수 없을 때 이 함수를 호출해도 됩니다. 이 함수가 NULL을 반환하면, 예외는 발생하지 않았고 호출자는 현재 스레드 상태를 사용할 수 없다고 가정해야 합니다.

int PyThreadState_SetAsyncExc (unsigned long *id*, *PyObject *exc*)

스레드에서 비동기적으로 예외를 발생시킵니다. *id* 인자는 대상 스레드의 스레드 id입니다; *exc*는 발생시킬 예외 객체입니다. 이 함수는 *exc*에 대한 어떤 참조도 훔치지 않습니다. 순진한 오용을 방지하려면, 이를 호출하는 자체 C 확장을 작성해야 합니다. GIL을 보유한 채로 호출해야 합니다. 수정된 스레드 상태를 반환합니다; 일반적으로 1이지만, 스레드 id를 찾지 못하면 0이 됩니다. *exc*가 NULL이면, 스레드에 대해 계류 중인 예외가 (있다면) 지워집니다. 이것은 예외를 일으키지 않습니다.

버전 3.7에서 변경: *id* 매개 변수의 형이 long에서 unsigned long으로 변경되었습니다.

void PyEval_AcquireThread (*PyThreadState *tstate*)

Acquire the global interpreter lock and set the current thread state to *tstate*, which should not be NULL. The lock must have been created earlier. If this thread already has the lock, deadlock ensues.

참고: 런타임이 파이널리제이션 될 때 스레드에서 이 함수를 호출하면, 스레드가 파이썬에 의해 만들어지지 않았더라도 스레드가 종료됩니다. 원치 않는 종료를 방지하려면 *_Py_IsFinalizing()*이나 *sys.is_finalizing()*을 사용하여 이 함수를 호출하기 전에 인터프리터가 파이널리제이션 되고 있는지 확인할 수 있습니다.

버전 3.8에서 변경: *PyEval_RestoreThread()*, *Py_END_ALLOW_THREADS()* 및 *PyGILState_Ensure()*와 일관되도록 갱신되었으며, 인터프리터가 파이널리제이션 하는 동안 호출되면 현재 스레드를 종료합니다.

*PyEval_RestoreThread()*는 (스레드가 초기화되지 않았을 때조차) 항상 사용할 수 있는 고수준 함수입니다.

void PyEval_ReleaseThread (*PyThreadState *tstate*)

현재 스레드 상태를 NULL로 재설정하고 전역 인터프리터 록을 해제합니다. 록은 이전에 만들어졌어야 하고 현재 스레드가 보유해야 합니다. NULL이 아니어야 하는 *tstate* 인자는 현재 스레드 상태를 나타내는 지 확인하는 데만 사용됩니다 — 그렇지 않으면, 치명적인 에러가 보고됩니다.

*PyEval_SaveThread()*는 (스레드가 초기화되지 않은 경우에조차) 항상 사용할 수 있는 고수준 함수입니다.

void PyEval_AcquireLock ()

전역 인터프리터 록을 획득합니다. 록은 이전에 만들어졌어야 합니다. 이 스레드에 이미 록이 있으면, 교착 상태가 발생합니다.

버전 3.2부터 폐지: 이 함수는 현재 스레드 상태를 갱신하지 않습니다. 대신 *PyEval_RestoreThread()*나 *PyEval_AcquireThread()*를 사용하십시오.

참고: 런타임이 파이널리제이션 될 때 스레드에서 이 함수를 호출하면, 스레드가 파이썬에 의해 만들어지지 않았더라도 스레드가 종료됩니다. 원치 않는 종료를 방지하려면 `_Py_IsFinalizing()` 이나 `sys.is_finalizing()` 을 사용하여 이 함수를 호출하기 전에 인터프리터가 파이널리제이션 되고 있는지 확인할 수 있습니다.

버전 3.8 에서 변경: `PyEval_RestoreThread()`, `Py_END_ALLOW_THREADS()` 및 `PyGILState_Ensure()` 와 일관되도록 갱신되었으며, 인터프리터가 파이널리제이션 하는 동안 호출되면 현재 스레드를 종료합니다.

void **PyEval_ReleaseLock** ()

전역 인터프리터 록을 해제합니다. 록은 이전에 만들어졌어야 합니다.

버전 3.2부터 폐지: 이 함수는 현재 스레드 상태를 갱신하지 않습니다. 대신 `PyEval_SaveThread()` 나 `PyEval_ReleaseThread()` 를 사용하십시오.

9.6 서브 인터프리터 지원

대부분의 경우, 단일 파이썬 인터프리터만 내장할 것입니다만, 같은 프로세스, 어쩌면 같은 스레드에서 여러 독립 인터프리터를 만들어야 하는 경우가 있습니다. 서브 인터프리터는 그렇게 할 수 있도록 합니다.

“메인” 인터프리터는 런타임이 초기화될 때 만들어지는 첫 번째 인터프리터입니다. 보통은 프로세스에서 유일한 파이썬 인터프리터입니다. 서브 인터프리터와 달리, 메인 인터프리터는 시그널 처리와 같은 고유한 프로세스 전역 책임을 갖습니다. 또한 런타임 초기화 동안 실행을 담당하며 일반적으로 런타임 파이널리제이션 동안 활성 인터프리터입니다. `PyInterpreterState_Main()` 함수는 그것의 상태에 대한 포인터를 반환합니다.

`PyThreadState_Swap()` 함수를 사용하여 서브 인터프리터 간에 전환할 수 있습니다. 다음 함수를 사용하여 만들고 파괴할 수 있습니다:

`PyThreadState*` **Py_NewInterpreter** ()

새 서브 인터프리터를 만듭니다. 이것은 파이썬 코드 실행을 위한 (거의) 완전히 분리된 환경입니다. 특히, 새 인터프리터에는 기본 모듈 `builtins`, `__main__` 및 `sys`를 포함하여, 모든 임포트된 모듈의 개별, 독립 버전을 갖습니다. 로드된 모듈 테이블(`sys.modules`)과 모듈 검색 경로(`sys.path`)도 별개입니다. 새 환경에는 `sys.argv` 변수가 없습니다. 새로운 표준 I/O 스트림 파일 객체 `sys.stdin`, `sys.stdout` 및 `sys.stderr`을 갖습니다 (단, 같은 하부 파일 기술자를 참조합니다).

반환 값은 새 서브 인터프리터에서 만들어진 첫 번째 스레드 상태를 가리킵니다. 이 스레드 상태는 현재 스레드 상태에서 만들어집니다. 실제 스레드가 만들어지지 않음에 유의하십시오; 아래 스레드 상태에 대한 설명을 참조하십시오. 새 인터프리터를 만드는 데 실패하면, `NULL`이 반환됩니다; 예외 상태는 현재 스레드 상태에 저장되고 현재 스레드 상태가 없을 수 있어서 예외가 설정되지 않습니다. (다른 모든 파이썬/C API 함수와 마찬가지로, 전역 인터프리터 록을 이 함수를 호출하기 전에 보유해야 하며 반환될 때 계속 유지됩니다; 그러나, 대부분의 다른 파이썬/C API 함수와 달리, 진입할 때 현재 스레드 상태가 있을 필요는 없습니다.)

확장 모듈은 다음과 같이 (서브) 인터프리터 간에 공유됩니다:

- 다단계 초기화를 사용하는 모듈의 경우, 예를 들어 `PyModule_FromDefAndSpec()`, 각 인터프리터에 대해 별도의 모듈 객체가 만들어지고 초기화됩니다. C 수준 정적과 전역 변수만 이러한 모듈 객체 간에 공유됩니다.
- 단단계 초기화를 사용하는 모듈의 경우, 예를 들어 `PyModule_Create()`, 특정 확장이 처음 임포트 될 때, 정상적으로 초기화되고, 모듈 디렉터리의 (얕은) 사본이 저장됩니다. 다른 (서브) 인터프리터가 같은 확장을 임포트 할 때, 새 모듈이 초기화되고 이 복사본의 내용으로 채워집니다; 확장의 `init` 함수는 호출되지 않습니다. 따라서 모듈 디렉터리의 객체는 (서브) 인터프리터 간에 공유되어, 원치 않는 동작을 일으킬 수 있습니다 (아래 **버그와 주의 사항**을 참조하십시오).

이것은 인터프리터가 `Py_FinalizeEx()`와 `Py_Initialize()`를 호출하여 완전히 다시 초기화된 후 확장을 임포트 할 때 일어나는 것과 다름에 유의하십시오; 이 경우, 확장의 `initmodule` 함수가 다시 호출됩니다. 단단계 초기화와 마찬가지로, 이는 C 수준의 정적과 전역 변수만 이러한 모듈 간에 공유됨을 의미합니다.

void **Py_EndInterpreter** (*PyThreadState* *tstate)

주어진 스레드 상태로 표현되는 (서브) 인터프리터를 파괴합니다. 주어진 스레드 상태는 현재 스레드 상태여야 합니다. 아래의 스레드 상태에 대한 설명을 참조하십시오. 호출이 반환되면, 현재 스레드 상태는 NULL입니다. 이 인터프리터와 관련된 모든 스레드 상태가 파괴됩니다. (전역 인터프리터 록을 이 함수를 호출하기 전에 보유해야 하며 반환될 때 여전히 유지됩니다.) `Py_FinalizeEx()`는 그 시점에서 명시적으로 파괴되지 않은 모든 서브 인터프리터를 파괴합니다.

9.6.1 버그와 주의 사항

서브 인터프리터(및 메인 인터프리터)는 같은 프로세스의 일부이기 때문에, 그들 간의 절연이 완벽하지 않습니다 — 예를 들어, `os.close()`와 같은 저수준 파일 연산을 사용하면 서로의 열린 파일에 (실수로 혹은 악의적으로) 영향을 미칠 수 있습니다. (서브) 인터프리터 간에 확장이 공유되는 방식 때문에, 일부 확장이 제대로 작동하지 않을 수 있습니다; 이것은 특히 단단계 초기화나 (정적) 전역 변수를 사용할 때 특히 그렇습니다. 한 서브 인터프리터에서 만든 객체를 다른 (서브) 인터프리터의 이름 공간에 삽입할 수 있습니다; 가능하면 피해야 합니다.

서브 인터프리터 간에 사용자 정의 함수, 메서드, 인스턴스 또는 클래스를 공유하지 않도록 특별한 주의를 기울여야 합니다. 이러한 객체에 의해 실행되는 임포트 연산은 잘못된 (서브) 인터프리터의 로드된 모듈 디렉터리에 영향을 미칠 수 있기 때문입니다. 위의 것들에서 접근할 수 있는 객체를 공유하지 않는 것도 마찬가지로 중요합니다.

또한 이 기능을 `PyGILState_*()` API와 결합하는 것은 까다로움에 유의하십시오. 이러한 API는, 서브 인터프리터의 존재로 인해 깨어진 가정인, 파이썬 스레드 상태와 OS 수준 스레드 사이의 일대일 관계를 가정하기 때문입니다. `PyGILState_Ensure()`와 `PyGILState_Release()` 호출의 일치하는 쌍 사이에 서브 인터프리터를 전환하지 않는 것이 좋습니다. 또한, 이러한 API를 사용하여 파이썬이 아닌 스레드에서 생성된 스레드에서 파이썬 코드를 호출할 수 있도록 하는 확장(가령 `ctypes`)은 서브 인터프리터를 사용할 때 망가질 수 있습니다.

9.7 비동기 알림

메인 인터프리터 스레드에 비동기 알림을 보내는 메커니즘이 제공됩니다. 이러한 알림은 함수 포인터와 void 포인터 인자의 형태를 취합니다.

int **Py_AddPendingCall** (int (*func)(void *), void *arg)

메인 인터프리터 스레드에서 호출할 함수를 예약합니다. 성공하면 0이 반환되고 `func`는 메인 스레드에서 호출되기 위해 큐에 추가됩니다. 실패 시, 예외 설정 없이 -1이 반환됩니다.

성공적으로 큐에 넣으면, `func`는 `arg` 인자를 사용하여 결국 메인 인터프리터 스레드에서 호출됩니다. 정상적으로 실행되는 파이썬 코드와 비교할 때 비동기적으로 호출되지만, 다음 두 조건이 모두 충족됩니다:

- 바이트 코드 경계에서;
- 메인 스레드가 전역 인터프리터 록을 보유하면서 (따라서 `func`는 전체 C API를 사용할 수 있습니다).

`func`는 성공하면 0을, 실패하면 예외 설정과 함께 -1을 반환해야 합니다. `func`는 다른 비동기 알림을 재귀적으로 수행하기 위해 중단되지 않지만, 전역 인터프리터 록이 해제되면 스레드를 전환하기 위해 여전히 중단될 수 있습니다.

이 함수는 실행하는 데 현재 스레드 상태가 필요하지 않으며, 전역 인터프리터 록이 필요하지 않습니다.

경고: 이것은 매우 특별한 경우에만 유용한, 저수준 함수입니다. *func*가 가능한 한 빨리 호출된다는 보장은 없습니다. 메인 스레드가 시스템 호출을 실행 중이라 바쁘면, 시스템 호출이 반환되기 전에 *func*가 호출되지 않습니다. 이 함수는 일반적으로 임의의 C 스레드에서 파이썬 코드를 호출하는 데 적합하지 않습니다. 대신, *PyGilState API*를 사용하십시오.

버전 3.1에 추가.

9.8 프로파일링과 추적

파이썬 인터프리터는 프로파일링과 실행 추적 기능을 연결하기 위한 몇 가지 저수준 지원을 제공합니다. 프로파일링, 디버깅 및 커버리지 분석 도구에 사용됩니다.

이 C 인터페이스를 사용하면 프로파일링이나 추적 코드가 파이썬 수준의 콜러블 객체를 통해 호출하는 오버헤드를 피하고, 대신 직접 C 함수를 호출할 수 있습니다. 시설의 필수 어트리뷰트는 변경되지 않았습니다; 인터페이스는 추적 함수를 스레드별로 설치할 수 있도록 하며, 추적 함수에 보고되는 기본 이벤트는 이전 버전의 파이썬 수준 추적 함수에 보고된 것과 같습니다.

int (*Py_tracefunc) (PyObject *obj, PyFrameObject *frame, int what, PyObject *arg)

PyEval_SetProfile() 과 *PyEval_SetTrace()* 를 사용하여 등록된 추적 함수의 형입니다. 첫 번째 매개 변수는 등록 함수에 *obj*로 전달된 객체이고, *frame*은 이벤트가 관련된 프레임 객체이고, *what*은 상수 *PyTrace_CALL*, *PyTrace_EXCEPTION*, *PyTrace_LINE*, *PyTrace_RETURN*, *PyTrace_C_CALL*, *PyTrace_C_EXCEPTION*, *PyTrace_C_RETURN* 또는 *PyTrace_OPCODE* 중 하나이고, *arg*는 *what*의 값에 따라 다릅니다:

<i>what</i> 의 값	<i>arg</i> 의 의미
<i>PyTrace_CALL</i>	항상 <i>Py_None</i> .
<i>PyTrace_EXCEPTION</i>	<i>sys.exc_info()</i> 에서 반환된 예외 정보.
<i>PyTrace_LINE</i>	항상 <i>Py_None</i> .
<i>PyTrace_RETURN</i>	호출자에게 반환되는 값, 또는 예외로 인한 것이면 <i>NULL</i> .
<i>PyTrace_C_CALL</i>	호출되는 함수 객체.
<i>PyTrace_C_EXCEPTION</i>	호출되는 함수 객체.
<i>PyTrace_C_RETURN</i>	호출되는 함수 객체.
<i>PyTrace_OPCODE</i>	항상 <i>Py_None</i> .

int PyTrace_CALL

함수나 메서드에 대한 새 호출이 보고되거나, 제너레이터에 대한 새 항목이 보고될 때 *Py_tracefunc* 함수에 대한 *what* 매개 변수의 값. 제너레이터 함수에 대한 이터레이터의 생성은 해당 프레임의 파이썬 바이트 코드로의 제어 전송이 없기 때문에 보고되지 않음에 유의하십시오.

int PyTrace_EXCEPTION

예외가 발생했을 때 *Py_tracefunc* 함수에 대한 *what* 매개 변수의 값. 콜백 함수는 실행되는 프레임 내에서 바이트 코드가 처리된 후 예외가 설정될 때 *what*에 대해 이 값으로 호출됩니다. 이것의 효과는 예외 전파로 인해 파이썬 스택이 되감기는 것입니다, 예외가 전파되어 각 프레임으로 반환할 때 콜백이 호출됩니다. 추적 함수만 이러한 이벤트를 수신합니다; 프로파일러에는 필요하지 않습니다.

int PyTrace_LINE

줄 번호 이벤트가 보고될 때 *what* 매개 변수로 *Py_tracefunc* 함수(하지만 프로파일링 함수는 아닙니다)에 전달되는 값. 해당 프레임의 *f_trace_lines*를 0으로 설정하여 해당 프레임에 대해 비활성화할 수 있습니다.

int PyTrace_RETURN

호출이 반환되려고 할 때 *Py_tracefunc* 함수에 대한 *what* 매개 변수의 값.

int PyTrace_C_CALL

C 함수가 호출되려고 할 때 *Py_tracefunc* 함수에 대한 *what* 매개 변수의 값.

int PyTrace_C_EXCEPTION

C 함수에서 예외가 발생했을 때 *Py_tracefunc* 함수에 대한 *what* 매개 변수의 값.

int PyTrace_C_RETURN

C 함수가 반환했을 때 *Py_tracefunc* 함수에 대한 *what* 매개 변수의 값.

int PyTrace_OPCODE

새 오퍼코드가 실행되려고 할 때 *Py_tracefunc* 함수(하지만 프로파일링 함수는 아닙니다)에 대한 *what* 매개 변수의 값. 이 이벤트는 기본적으로 방출되지 않습니다: 프레임의 *f_trace_opcodes*를 1로 설정하여 명시적으로 요청해야 합니다.

void PyEval_SetProfile (*Py_tracefunc func*, *PyObject *obj*)

프로파일러 함수를 *func*로 설정합니다. *obj* 매개 변수는 첫 번째 매개 변수로 함수에 전달되며, 임의의 파이썬 객체나 NULL일 수 있습니다. 프로파일 함수가 상태를 유지해야 하면, 스레드마다 *obj*에 다른 값을 사용하면 저장하기에 편리하고 스레드 안전한 위치를 제공합니다. 프로파일 함수는 PyTrace_LINE, PyTrace_OPCODE 및 PyTrace_EXCEPTION을 제외한 모든 관찰되는 이벤트에 대해 호출됩니다.

void PyEval_SetTrace (*Py_tracefunc func*, *PyObject *obj*)

추적 함수를 *func*로 설정합니다. 추적 함수가 줄 번호 이벤트와 오퍼코드별 이벤트를 수신하지만, 호출되는 C 함수 객체와 관련된 이벤트를 수신하지 않는다는 점을 제외하면, *PyEval_SetProfile()*과 유사합니다. *PyEval_SetTrace()*를 사용하여 등록된 모든 추적 함수는 *what* 매개 변수의 값으로 PyTrace_C_CALL, PyTrace_C_EXCEPTION 또는 PyTrace_C_RETURN을 수신하지 않습니다.

9.9 고급 디버거 지원

이 함수들은 고급 디버깅 도구에서만 사용하기 위한 것입니다.

*PyInterpreterState** **PyInterpreterState_Head** ()

인터프리터 상태 객체들의 리스트의 머리에 있는 객체를 반환합니다.

*PyInterpreterState** **PyInterpreterState_Main** ()

메인 인터프리터 상태 객체를 반환합니다.

*PyInterpreterState** **PyInterpreterState_Next** (*PyInterpreterState *interp*)

인터프리터 상태 객체들의 리스트에서 *interp* 이후의 다음 인터프리터 상태 객체를 반환합니다.

*PyThreadState** **PyInterpreterState_ThreadHead** (*PyInterpreterState *interp*)

인터프리터 *interp*와 관련된 스레드 리스트에서 첫 번째 *PyThreadState* 객체에 대한 포인터를 반환합니다.

*PyThreadState** **PyThreadState_Next** (*PyThreadState *tstate*)

같은 *PyInterpreterState* 객체에 속하는 모든 스레드 객체 리스트에서 *tstate* 이후의 다음 스레드 상태 객체를 반환합니다.

9.10 스레드 로컬 저장소 지원

파이썬 인터프리터는 파이썬 수준의 스레드 로컬 저장소 API(`threading.local`)를 지원하기 위해 하부 네이티브 TLS 구현을 래핑하는 스레드 로컬 저장소(TLS)에 대한 저수준 지원을 제공합니다. CPython C 수준 API는 `pthread`와 윈도우에서 제공하는 API와 유사합니다: 스레드 키와 함수를 사용하여 스레드 당 `void*` 값을 연결합니다.

이러한 함수를 호출할 때 GIL을 보유할 필요는 없습니다; 그들은 자체 록을 제공합니다.

Python.h에는 TLS API 선언이 포함되어 있지 않음에 유의하십시오, 스레드 로컬 저장소를 사용하려면 `pthread.h`를 포함해야 합니다.

참고: 이러한 API 함수 중 어느 것도 `void*` 값을 대신해서 메모리 관리를 처리하지 않습니다. 직접 할당하고 할당 해제해야 합니다. `void*` 값이 `PyObject*` 이라면, 이 함수들은 참조 횟수 연산도 수행하지 않습니다.

9.10.1 스레드별 저장소 (TSS - Thread Specific Storage) API

CPython 인터프리터 내에서 기존 TLS API의 사용을 대체하기 위해 TSS API가 도입되었습니다. 이 API는 스레드 키를 나타내기 위해 `int` 대신 새로운 형 `Py_tss_t`를 사용합니다.

버전 3.7에 추가.

더 보기:

“CPython의 스레드-로컬 저장소를 위한 새로운 C-API” (PEP 539)

`Py_tss_t`

이 자료 구조는 스레드 키의 상태를 나타내며, 정의는 하부 TLS 구현에 따라 달라질 수 있으며, 키의 초기화 상태를 나타내는 내부 필드가 있습니다. 이 구조체에는 공개 멤버가 없습니다.

`Py_LIMITED_API`가 정의되지 않을 때, `Py_tss_NEEDS_INIT`로 이 형의 정적 할당이 허용됩니다.

`Py_tss_NEEDS_INIT`

이 매크로는 `Py_tss_t` 변수의 초기화자(initializer)로 확장됩니다. 이 매크로는 `Py_LIMITED_API`에서 정의되지 않음에 유의하십시오.

동적 할당

`Py_LIMITED_API`로 빌드된 확장 모듈에 필요한, 빌드 시점에 구현이 불투명해서 형의 정적 할당이 불가능한 `Py_tss_t`의 동적 할당.

`Py_tss_t*` `PyThread_tss_alloc()`

`Py_tss_NEEDS_INIT`로 초기화된 값과 같은 상태의 값을 반환하거나, 동적 할당 실패 시 `NULL`을 반환합니다.

`void PyThread_tss_free(Py_tss_t *key)`

모든 관련 스레드 로컬의 대입을 해제하도록 `PyThread_tss_delete()`를 먼저 호출한 후, `PyThread_tss_alloc()`에 의해 할당된 주어진 `key`를 할당 해제합니다. `key` 인자가 `NULL`이면 아무런 일도 하지 않습니다.

참고: 해제된 키는 매달린(dangling) 포인터가 됩니다, 키를 `NULL`로 재설정해야 합니다.

메서드

이 함수들의 매개 변수 `key`는 `NULL`이 아니어야 합니다. 또한, 주어진 `Py_tss_t`가 `PyThread_tss_create()`로 초기화되지 않았으면, `PyThread_tss_set()`과 `PyThread_tss_get()`의 동작은 정의되지 않습니다.

`int PyThread_tss_is_created(Py_tss_t *key)`

주어진 `Py_tss_t`가 `PyThread_tss_create()`로 초기화되었으면 0이 아닌 값을 반환합니다.

`int PyThread_tss_create(Py_tss_t *key)`

TSS 키 초기화에 성공하면 0 값을 반환합니다. `key` 인자가 가리키는 값이 `Py_tss_NEEDS_INIT`로 초기화되지 않으면 동작이 정의되지 않습니다. 이 함수는 같은 키에서 반복적으로 호출될 수 있습니다 – 이미 초기화된 키에 대해 호출하면 아무런 일도 하지 않으며 즉시 성공을 반환합니다.

`void PyThread_tss_delete(Py_tss_t *key)`

TSS 키를 삭제하여 모든 스레드에서 키와 관련된 값을 잊게 하고, 키의 초기화 상태를 초기화되지 않음으로 변경합니다. 파괴된 키는 `PyThread_tss_create()`로 다시 초기화할 수 있습니다. 이 함수는 같은 키에서 반복적으로 호출될 수 있습니다 – 이미 파괴된 키에 대해 호출하면 아무런 일도 하지 않습니다.

`int PyThread_tss_set(Py_tss_t *key, void *value)`

현재 스레드에서 `void*` 값을 TSS 키와 성공적으로 연결했음을 나타내는 0 값을 반환합니다. 각 스레드에는 키에서 `void*` 값으로의 고유한 매핑이 있습니다.

`void* PyThread_tss_get(Py_tss_t *key)`

현재 스레드의 TSS 키와 관련된 `void*` 값을 반환합니다. 현재 스레드에 키와 연결된 값이 없으면 `NULL`을 반환합니다.

9.10.2 스레드 로컬 저장소 (TLS) API

버전 3.7부터 폐지: 이 API는 스레드별 저장소 (TSS) API로 대체됩니다.

참고: 이 버전의 API는 `int`로 안전하게 캐스트 할 수 없는 방식으로 네이티브 TLS 키가 정의된 플랫폼을 지원하지 않습니다. 이러한 플랫폼에서, `PyThread_create_key()`는 실패 상태로 즉시 반환되며, 다른 TLS 함수는 이러한 플랫폼에서 모두 아무런 일도 하지 않습니다.

위에서 언급한 호환성 문제로 인해, 이 버전의 API를 새 코드에서 사용해서는 안 됩니다.

`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()`

CHAPTER 10

파이썬 초기화 구성

버전 3.8에 추가.

구조체:

- *PyConfig*
- *PyPreConfig*
- *PyStatus*
- *PyWideStringList*

함수:

- *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()`

사전 구성(`PyPreConfig` 형)은 `_PyRuntime.preconfig`에 저장되고 구성(`PyConfig` 형)은 `PyInterpreterState.config`에 저장됩니다.

초기화, 파이널리제이션 및 스레드도 참조하십시오.

더 보기:

PEP 587 “파이썬 초기화 구성”.

10.1 PyWideStringList

PyWideStringList

`wchar_t*` 문자열의 리스트.

`length`가 0이 아니면, `items`는 `NULL`이 아니어야 하고 모든 문자열은 `NULL`이 아니어야 합니다.

메서드:

***PyStatus* `PyWideStringList_Append`** (*`PyWideStringList`* *list, const `wchar_t` *item)

*item*을 *list*에 추가합니다.

이 함수를 호출하려면 파이썬을 사전 초기화해야 합니다.

***PyStatus* `PyWideStringList_Insert`** (*`PyWideStringList`* *list, `Py_ssize_t` index, const `wchar_t` *item)

*item*을 *list*의 *index*에 삽입합니다.

*index*가 *list* 길이보다 크거나 같으면, *item*을 *list*에 추가(append) 합니다.

*index*는 0보다 크거나 같아야 합니다.

이 함수를 호출하려면 파이썬을 사전 초기화해야 합니다.

구조체 필드:

`Py_ssize_t` `length`

리스트 길이.

`wchar_t` `items`**

리스트 항목들.

10.2 PyStatus

PyStatus

초기화 함수 상태를 저장하는 구조체: 성공, 에러 또는 종료.

에러의 경우, 에러를 만든 C 함수 이름을 저장할 수 있습니다.

구조체 필드:

int exitcode

종료 코드. `exit()`에 전달된 인자.

const char *err_msg

에러 메시지.

const char *func

에러를 만든 함수의 이름, NULL일 수 있습니다.

상태를 만드는 함수:

PyStatus **PyStatus_Ok** (void)

성공.

PyStatus **PyStatus_Error** (const char *err_msg)

메시지가 포함된 초기화 에러.

PyStatus **PyStatus_NoMemory** (void)

메모리 할당 실패 (메모리 부족).

PyStatus **PyStatus_Exit** (int exitcode)

지정된 종료 코드로 파이썬을 종료합니다.

상태를 처리하는 함수:

int **PyStatus_Exception** (*PyStatus* status)

상태가 에러입니까? 아니면 종료입니까? 참이면, 예외를 처리해야 합니다; 예를 들어 `Py_ExitStatusException()`을 호출하여.

int **PyStatus_IsError** (*PyStatus* status)

결과가 에러입니까?

int **PyStatus_IsExit** (*PyStatus* status)

결과가 종료입니까?

void **Py_ExitStatusException** (*PyStatus* status)

*status*가 종료이면 `exit(exitcode)`를 호출합니다. *status*가 에러이면 에러 메시지를 인쇄하고 0이 아닌 종료 코드로 종료합니다. `PyStatus_Exception(status)`가 0이 아닐 때만 호출해야 합니다.

참고: 내부적으로, 파이썬은 `PyStatus.func`를 설정하는 데는 매크로를 사용하는 반면, `func`가 NULL로 설정된 상태를 만드는 데는 함수를 사용합니다.

예:

```
PyStatus alloc(void **ptr, size_t size)
{
    *ptr = PyMem_RawMalloc(size);
    if (*ptr == NULL) {
        return PyStatus_NoMemory();
    }
}
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```

    return PyStatus_Ok();
}

int main(int argc, char **argv)
{
    void *ptr;
    PyStatus status = alloc(&ptr, 16);
    if (PyStatus_Exception(status)) {
        Py_ExitStatusException(status);
    }
    PyMem_Free(ptr);
    return 0;
}

```

10.3 PyPreConfig

PyPreConfig

파이썬을 사전 초기화하는 데 사용되는 구조체:

- 파이썬 메모리 할당자를 설정합니다
- LC_CTYPE 로케일을 구성합니다
- UTF-8 모드를 설정합니다

사전 구성을 초기화하는 함수:

void **PyPreConfig_InitPythonConfig** (*PyPreConfig *preconfig*)

파이썬 구성으로 사전 구성을 초기화합니다.

void **PyPreConfig_InitIsolatedConfig** (*PyPreConfig *preconfig*)

격리된 구성으로 사전 구성을 초기화합니다.

구조체 필드:

int **allocator**

메모리 할당자의 이름:

- PYMEM_ALLOCATOR_NOT_SET (0): 메모리 할당자를 변경하지 않습니다(기본값을 사용합니다)
- PYMEM_ALLOCATOR_DEFAULT (1): 기본 메모리 할당자
- PYMEM_ALLOCATOR_DEBUG (2): 디버그 훅이 있는 기본 메모리 할당자
- PYMEM_ALLOCATOR_MALLOC (3): malloc()의 사용을 강제합니다
- PYMEM_ALLOCATOR_MALLOC_DEBUG (4): 디버그 훅이 있는 malloc()의 사용을 강제합니다
- PYMEM_ALLOCATOR_PYMALLOC (5): 파이썬 *pymalloc* 메모리 할당자
- PYMEM_ALLOCATOR_PYMALLOC_DEBUG (6): 디버그 훅이 있는 파이썬 *pymalloc* 메모리 할당자

파이썬이 `--without-pymalloc`을 사용하여 구성되면 PYMEM_ALLOCATOR_PYMALLOC과 PYMEM_ALLOCATOR_PYMALLOC_DEBUG은 지원되지 않습니다

메모리 관리를 참조하십시오.

int **configure_locale**

LC_CTYPE 로케일을 사용자 선호 로케일로 설정합니까?? 0과 같으면, `coerce_c_locale`과 `coerce_c_locale_warn`을 0으로 설정합니다.

int `coerce_c_locale`
2와 같으면, C 로케일을 강제합니다; 1과 같으면, LC_CTYPE 로케일을 읽고 강제할지 결정합니다.

int `coerce_c_locale_warn`
0이 아니면, C 로케일이 강제될 때 경고가 발생합니다.

int `dev_mode`
`PyConfig.dev_mode`를 참조하십시오.

int `isolated`
`PyConfig.isolated`를 참조하십시오.

int `legacy_windows_fs_encoding` (*Windows only*)
0이 아니면, UTF-8 모드를 비활성화하고, 파이썬 파일 시스템 인코딩을 mbcs로 설정하고, 파일 시스템 에러 처리기를 replace로 설정합니다.

윈도우에서만 사용 가능합니다. `#ifdef MS_WINDOWS` 매크로는 윈도우 특정 코드에 사용할 수 있습니다.

int `parse_argv`
0이 아니면, `Py_PreInitializeFromArgs()`와 `Py_PreInitializeFromBytesArgs()`는 일반 파이썬이 명령 줄 인자를 구문 분석하는 것과 같은 방식으로 argv 인자를 구문 분석합니다. 명령 줄 인자를 참조하십시오.

int `use_environment`
`PyConfig.use_environment`를 참조하십시오.

int `utf8_mode`
0이 아니면, UTF-8 모드를 활성화합니다.

10.4 PyPreConfig를 사용한 사전 초기화

파이썬을 사전 초기화하는 함수:

PyStatus `Py_PreInitialize` (const *PyPreConfig* *preconfig)
preconfig 사전 구성에서 파이썬을 사전 초기화합니다.

PyStatus `Py_PreInitializeFromBytesArgs` (const *PyPreConfig* *preconfig, int argc, char * const *argv)
preconfig 사전 구성과 명령 줄 인자(바이트 문자열)에서 파이썬을 사전 초기화합니다.

PyStatus `Py_PreInitializeFromArgs` (const *PyPreConfig* *preconfig, int argc, wchar_t * const *argv)
preconfig 사전 구성과 명령 줄 인자(와이드 문자열)에서 파이썬을 사전 초기화합니다.

호출자는 `PyStatus_Exception()`과 `Py_ExitStatusException()`을 사용하여 예외(예러나 종료)를 처리해야 합니다.

파이썬 구성(`PyPreConfig_InitPythonConfig()`)의 경우, 명령 줄 인자로 파이썬을 초기화하면, 인코딩과 같은 사전 구성에 영향을 주기 때문에, 파이썬을 사전 구성하기 위해 명령 줄 인자도 전달되어야 합니다. 예를 들어, `-X utf8` 명령 줄 옵션은 UTF-8 모드를 활성화합니다.

`PyMem_SetAllocator()`는 `Py_PreInitialize()` 이후에 `Py_InitializeFromConfig()` 이전에 호출하여 사용자 정의 메모리 할당자를 설치할 수 있습니다. `PyPreConfig.allocator`가 `PYMEM_ALLOCATOR_NOT_SET`으로 설정되면 `Py_PreInitialize()` 전에 호출할 수 있습니다.

`PyMem_RawMalloc()`과 같은 파이썬 메모리 할당 함수는 파이썬 사전 초기화 전에 사용해서는 안 되지만, `malloc()`과 `free()`를 직접 호출하는 것은 항상 안전합니다. 사전 초기화 전에 `Py_DecodeLocale()`을 호출하면 안 됩니다.

UTF-8 모드를 활성화하기 위해 사전 초기화를 사용하는 예:

```

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 */

Py_Initialize();
/* ... use Python API here ... */
Py_Finalize();

```

10.5 PyConfig

PyConfig

파이썬을 구성하기 위한 대부분의 파라미터를 포함하는 구조체.

구조체 메서드:

void PyConfig_InitPythonConfig (PyConfig *config)

파이썬 구성으로 구성을 초기화합니다.

void PyConfig_InitIsolatedConfig (PyConfig *config)

격리된 구성으로 구성을 초기화합니다.

PyStatus PyConfig_SetString (PyConfig *config, wchar_t * const *config_str, const wchar_t *str)

와이드 문자열 *str*을 *config_str로 복사합니다.

필요하면 파이썬을 사전 초기화합니다.

PyStatus PyConfig_SetBytesString (PyConfig *config, wchar_t * const *config_str, const char *str)

Py_DecodeLocale()을 사용하여 *str*을 디코딩하고 결과를 *config_str에 설정합니다.

필요하면 파이썬을 사전 초기화합니다.

PyStatus PyConfig_SetArgv (PyConfig *config, int argc, wchar_t * const *argv)

와이드 문자열로 명령 줄 인자를 설정합니다.

필요하면 파이썬을 사전 초기화합니다.

PyStatus PyConfig_SetBytesArgv (PyConfig *config, int argc, char * const *argv)

명령 줄 인자를 설정합니다: Py_DecodeLocale()를 사용하여 바이트열을 디코딩합니다.

필요하면 파이썬을 사전 초기화합니다.

PyStatus PyConfig_SetWideStringList (PyConfig *config, PyWideStringList *list, Py_ssize_t length, wchar_t **items)

와이드 문자열 리스트 *list*를 *length*와 *items*로 설정합니다.

필요하면 파이썬을 사전 초기화합니다.

PyStatus PyConfig_Read (PyConfig *config)

모든 파이썬 구성을 읽습니다.

이미 초기화된 필드는 변경되지 않습니다.

필요하면 파이썬을 사전 초기화합니다.

void **PyConfig_Clear** (*PyConfig *config*)

구성 메모리를 해제합니다.

대부분의 PyConfig 메서드는 필요하면 파이썬을 사전 초기화합니다. 이 경우, *PyConfig*를 기반으로 하는 파이썬 사전 초기화 구성입니다. *PyPreConfig*와 공통인 구성 필드가 조정되면, *PyConfig* 메서드를 호출하기 전에 설정해야 합니다:

- *dev_mode*
- *isolated*
- *parse_argv*
- *use_environment*

또한, *PyConfig_SetArgv()* 나 *PyConfig_SetBytesArgv()* 가 사용되면, 사전 초기화 구성이 명령 줄 인자 (parse_argv가 0이 아니면)에 의존하기 때문에 다른 메서드보다 먼저 이 메서드를 호출해야 합니다.

이 메서드의 호출자는 *PyStatus_Exception()* 과 *Py_ExitStatusException()* 을 사용하여 예외 (예러나 종료)를 처리해야 합니다.

구조체 필드:

PyWideStringList **argv**

명령 줄 인자, *sys.argv*. 일반 파이썬이 파이썬 명령 줄 인자를 구문 분석하는 것과 같은 방식으로 *argv*를 구문 분석하려면 *parse_argv*를 참조하십시오. *argv*가 비어 있으면, 빈 문자열이 추가 되어 *sys.argv*가 항상 존재하고 절대로 비어 있지 않도록 합니다.

wchar_t* **base_exec_prefix**

sys.base_exec_prefix.

wchar_t* **base_executable**

sys._base_executable: `__PYENVV_LAUNCHER__` 환경 변수 값, 또는 *PyConfig.executable*의 사본.

wchar_t* **base_prefix**

sys.base_prefix.

int **buffered_stdio**

0과 같으면, 버퍼링 되지 않는 모드를 활성화하여, *stdout*과 *stderr* 스트림을 버퍼링하지 않도록 합니다.

*stdin*은 항상 버퍼링 모드로 열립니다.

int **bytes_warning**

1과 같으면, *bytes*나 *bytearray*를 *str*와 비교하거나, 또는 *bytes*를 *int*와 비교할 때 경고를 발행합니다. 2 이상이면, *BytesWarning* 예외를 발생시킵니다.

wchar_t* **check_hash_pycs_mode**

해시 기반 .pyc 파일의 유효성 검증 동작을 제어합니다 (**PEP 552**를 참조하십시오): `--check-hash-based-pycs` 명령 줄 옵션 값.

유효한 값: *always*, *never* 및 *default*.

기본값은 *default*입니다.

int **configure_c_stdio**

0이 아니면, C 표준 스트림(*stdio*, *stdout*, *stderr*)을 구성합니다. 예를 들어, 윈도우에서 해당 모드를 *O_BINARY*로 설정합니다.

int dev_mode

개발 모드: `-X dev`를 참조하십시오.

int dump_refs

0이 아니면, 종료 시 여전히 활성 상태인 모든 객체를 덤프합니다.

파이썬의 디버그 빌드가 필요합니다 (`Py_REF_DEBUG` 매크로를 정의해야 합니다).

wchar_t* exec_prefix

`sys.exec_prefix`.

wchar_t* executable

`sys.executable`.

int faulthandler

0이 아니면, 시작 시 `faulthandler.enable()` 을 호출합니다.

wchar_t* filesystem_encoding

파일 시스템 인코딩, `sys.getfilesystemencoding()`.

wchar_t* filesystem_errors

파일 시스템 인코딩 에러, `sys.getfilesystemencodeerrors()`.

unsigned long hash_seed

int use_hash_seed

무작위 해시 함수 시드.

`use_hash_seed`가 0이면, 파이썬 시작 시 시드가 무작위로 선택되고, `hash_seed`는 무시됩니다.

wchar_t* home

파이썬 홈 디렉터리.

기본적으로 `PYTHONHOME` 환경 변수값에서 초기화됩니다.

int import_time

0이 아니면, импорт 시간을 프로파일 합니다.

int inspect

스크립트나 명령을 실행한 후 대화식 모드로 들어갑니다.

int install_signal_handlers

시그널 처리기를 설치합니까?

int interactive

대화식 모드.

int isolated

0보다 크면, 격리 모드를 활성화합니다:

- `sys.path`에는 스크립트 디렉터리(`argv[0]` 이나 현재 디렉터리에서 계산됩니다)도 사용자의 `site-packages` 디렉터리도 없습니다.
- 파이썬 REPL은 대화식 프롬프트에서 `readline`을 импорт 하지도 기본 `readline` 구성을 활성화 하지도 않습니다.
- `use_environment`와 `user_site_directory`를 0으로 설정합니다.

int legacy_windows_stdio

0이 아니면, `sys.stdin`, `sys.stdout` 및 `sys.stderr`에 `io.WindowsConsoleIO` 대신 `io.FileIO`를 사용합니다.

윈도우에서만 사용 가능합니다. `#ifdef MS_WINDOWS` 매크로는 윈도우 특정 코드에 사용할 수 있습니다.

int malloc_stats

0이 아니면, 종료 시 파이썬 *pymalloc* 메모리 할당자에 대한 통계를 덤프합니다.

파이썬이 `--without-pymalloc`을 사용하여 빌드되면 이 옵션은 무시됩니다.

wchar_t* pythonpath_env

`DELIM(os.path.pathsep)`으로 구분된 문자열로 표현된 모듈 검색 경로.

기본적으로 `PYTHONPATH` 환경 변수값에서 초기화됩니다.

PyWideStringList module_search_paths**int module_search_paths_set**

`sys.path.module_search_paths_set`이 0과 같으면, 경로 구성을 계산하는 함수가 `module_search_paths`를 재정의합니다.

int optimization_level

컴파일 최적화 수준:

- 0: 틸 구멍 최적화기(Peephole optimizer) (그리고 `__debug__`이 `True`로 설정됩니다)
- 1: 어서션을 제거합니다, `__debug__`을 `False`로 설정합니다
- 2: 독스트링을 제거합니다

int parse_argv

0이 아니면, 일반 파이썬 명령 줄 인자와 같은 방식으로 *argv*를 구문 분석하고, *argv*에서 파이썬 인자를 제거합니다: 명령 줄 인자를 참조하십시오.

int parser_debug

0이 아니면, 구문 분석기 디버깅 출력을 켭니다 (컴파일 옵션에 따라, 전문가용입니다).

int pathconfig_warnings

0과 같으면, 경로 구성을 계산할 때 경고를 억제합니다 (유닉스 전용, 윈도우는 아무런 경고도 로그하지 않습니다). 그렇지 않으면, 경고가 `stderr`에 기록됩니다.

wchar_t* prefix

`sys.prefix`.

wchar_t* program_name

프로그램 이름. *executable*을 초기화하는 데 사용되며, 초기 에러 메시지에서도 사용됩니다.

wchar_t* pycache_prefix

`sys.pycache_prefix`: `.pyc` 캐시 접두사.

`NULL`이면, `sys.pycache_prefix`는 `None`으로 설정됩니다.

int quiet

침묵 모드. 예를 들어, 대화식 모드에서 저작권과 버전 메시지를 표시하지 않습니다.

wchar_t* run_command

`python3 -c COMMAND` 인자. *Py_RunMain()*에서 사용합니다.

wchar_t* run_filename

`python3 FILENAME` 인자. *Py_RunMain()*에서 사용합니다.

wchar_t* run_module

`python3 -m MODULE` 인자. *Py_RunMain()*에서 사용합니다.

int show_alloc_count

종료 시 할당 횟수를 표시합니까?

`-X showalloccount` 명령 줄 옵션으로 1로 설정됩니다.

`COUNT_ALLOCS` 매크로가 정의된 특수한 파이썬 빌드가 필요합니다.

int show_ref_count

종료 시 총 참조 횟수를 표시합니까?

-X showrefcount 명령 줄 옵션으로 1로 설정됩니다.

파이썬의 디버그 빌드가 필요합니다(Py_REF_DEBUG 매크로를 정의해야 합니다).

int site_import

시작할 때 site 모듈을 임포트 합니까?

int skip_source_first_line

소스의 첫 줄을 건너뛰니까?

wchar_t* stdio_encoding

wchar_t* stdio_errors

sys.stdin, sys.stdout 및 sys.stderr의 인코딩과 인코딩 에러.

int tracemalloc

0이 아니면, 시작 시 tracemalloc.start()를 호출합니다.

int use_environment

0보다 크면, 환경 변수를 사용합니다.

int user_site_directory

0이 아니면, 사용자 사이트 디렉터리를 sys.path에 추가합니다.

int verbose

0이 아니면, 상세 모드를 활성화합니다.

PyWideStringList warnoptions

sys.warnoptions: 경고 필터를 빌드하기 위한 warnings 모듈의 옵션: 가장 낮은 것에서 가장 높은 우선순위로.

warnings 모듈은 sys.warnoptions를 역순으로 추가합니다: 마지막 *PyConfig.warnoptions* 항목은 가장 먼저 검사되는 warnings.filters의 첫 번째 항목이 됩니다(가장 높은 우선순위).

int write_bytecode

0이 아니면, .pyc 파일을 기록합니다.

sys.dont_write_bytecode는 *write_bytecode*의 반전된 값으로 초기화됩니다.

PyWideStringList xoptions

sys._xoptions.

parse_argv가 0이 아니면, 일반 파이썬이 명령 줄 인자를 구문 분석하는 것과 같은 방식으로 argv 인자가 구문 분석되고, argv에서 파이썬 인자가 제거됩니다: 명령 줄 인자를 참조하십시오.

xoptions 옵션은 다른 옵션을 설정하기 위해 구문 분석됩니다: -X 옵션을 참조하십시오.

10.6 PyConfig를 사용한 초기화

파이썬을 초기화하는 함수:

PyStatus Py_InitializeFromConfig (const *PyConfig* *config)

config 구성에서 파이썬을 초기화합니다.

호출자는 *PyStatus_Exception()*과 *Py_ExitStatusException()*을 사용하여 예외(에러나 종료)를 처리해야 합니다.

`PyImport_FrozenModules`, `PyImport_AppendInittab()` 또는 `PyImport_ExtendInittab()` 을 사용하면, 파이썬 사전 초기화 후에 그리고 파이썬 초기화 전에 설정하거나 호출해야 합니다.

프로그램 이름을 설정하는 예:

```
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);
    return;

fail:
    PyConfig_Clear(&config);
    Py_ExitStatusException(status);
}
```

기본 구성을 수정하는 더 완전한 예, 구성을 읽은 다음 일부 파라미터를 대체합니다:

```
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,
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```

                                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 격리된 구성

*PyPreConfig_InitIsolatedConfig()*와 *PyConfig_InitIsolatedConfig()* 함수는 시스템에서 파이썬을 격리하는 구성을 만듭니다. 예를 들어, 파이썬을 응용 프로그램에 내장하기 위해.

이 구성은 전역 구성 변수, 환경 변수, 명령 줄 인자(*PyConfig.argv*가 구문 분석되지 않습니다) 및 사용자 사이트 디렉터리를 무시합니다. C 표준 스트림(예를 들어 `stdout`)과 `LC_CTYPE` 로케일은 변경되지 않습니다. 시그널 처리기가 설치되지 않습니다.

구성 파일은 여전히 이 구성에 사용됩니다. 이러한 구성 파일을 무시하고 기본 경로 구성을 계산하는 함수를 피하려면 **경로 구성**(“출력 필드”)을 설정하십시오.

10.8 파이썬 구성

*PyPreConfig_InitPythonConfig()*와 *PyConfig_InitPythonConfig()* 함수는 일반 파이썬처럼 동작하는 사용자 정의된 파이썬을 빌드하기 위한 구성을 만듭니다.

환경 변수와 명령 줄 인자는 파이썬을 구성하는 데 사용되는 반면, 전역 구성 변수는 무시됩니다.

이 함수는 `LC_CTYPE` 로케일, `PYTHONUTF8` 및 `PYTHONCOERCECLOCALE` 환경 변수에 따라 C 로케일 강제(**PEP 538**)와 UTF-8 모드(**PEP 540**)를 활성화합니다.

항상 격리 모드에서 실행되는 사용자 정의 파이썬의 예:

```

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). */
}

```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```

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 경로 구성

`PyConfig`에는 경로 구성을 위한 여러 필드가 포함되어 있습니다:

- 경로 구성 입력:
 - `PyConfig.home`
 - `PyConfig.pathconfig_warnings`
 - `PyConfig.program_name`
 - `PyConfig.pythonpath_env`
 - 현재 작업 디렉터리: 절대 경로를 얻기 위해
 - (`PyConfig.program_name`에서) 프로그램 전체 경로를 얻기 위한 PATH 환경 변수
 - `__PYENV_LAUNCHER__` 환경 변수
 - (윈도 전용) `HKEY_CURRENT_USER` 와 `HKEY_LOCAL_MACHINE` 의 “SoftwarePythonPythonCoreX.YPythonPath” 아래에 있는 레지스트리의 응용 프로그램 경로 (여기서 X.Y는 파이썬 버전입니다).
- 경로 구성 출력 필드:
 - `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`

적어도 하나의 “출력 필드”가 설정되어 있지 않으면, 파이썬은 설정되지 않은 필드를 채우기 위해 경로 구성을 계산합니다. `module_search_paths_set`이 0이면, `module_search_paths`가 재정의되고 `module_search_paths_set`이 1로 설정됩니다.

위에 나열된 모든 경로 구성 출력 필드를 명시적으로 설정하여 기본 경로 구성을 계산하는 함수를 완전히 무시할 수 있습니다. 비어 있지 않아도 문자열은 설정된 것으로 간주합니다. `module_search_paths_set`이 1로 설정되면 `module_search_paths`는 설정된 것으로 간주합니다. 이 경우, 경로 구성 입력 필드도 무시됩니다.

경로 구성을 계산할 때 경고를 억제하려면 `pathconfig_warnings`를 0으로 설정하십시오 (유닉스 전용, 윈도우는 어떤 경고도 로그 하지 않습니다).

`base_prefix`나 `base_exec_prefix` 필드가 설정되지 않으면, 각각 `prefix`와 `exec_prefix`의 값을 상속합니다.

`Py_RunMain()`과 `Py_Main()`은 `sys.path`를 수정합니다:

- `run_filename`이 설정되고 `__main__.py` 스크립트를 포함하는 디렉터리이면, `run_filename`을 `sys.path` 앞에 추가합니다.
- `isolated`가 0이면:
 - `run_module`이 설정되면, 현재 디렉터리를 `sys.path` 앞에 추가합니다. 현재 디렉터리를 읽을 수 없으면 아무것도 하지 않습니다.
 - `run_filename`이 설정되면, 파일명의 디렉터리를 `sys.path` 앞에 추가합니다.
 - 그렇지 않으면, 빈 문자열을 `sys.path` 앞에 추가합니다.

`site_import`가 0이 아니면, `site` 모듈이 `sys.path`를 수정할 수 있습니다. `user_site_directory`가 0이 아니고 사용자의 `site-package` 디렉터리가 존재하면, `site` 모듈은 사용자의 `site-package` 디렉터리를 `sys.path`에 추가합니다.

다음과 같은 구성 파일이 경로 구성에 사용됩니다:

- `pyvenv.cfg`
- `python._pth` (윈도우 전용)
- `pybuilddir.txt` (유닉스 전용)

`__PYENV_LAUNCHER__` 환경 변수는 `PyConfig.base_executable`을 설정하는 데 사용됩니다

10.10 `Py_RunMain()`

`int Py_RunMain (void)`

명령 줄이나 구성에서 지정된 명령 (`PyConfig.run_command`), 스크립트 (`PyConfig.run_filename`) 또는 모듈 (`PyConfig.run_module`)을 실행합니다.

기본적으로, 그리고 `-i` 옵션을 사용할 때, REPL을 실행합니다.

마지막으로, 파이썬을 파이널라이즈 하고 `exit()` 함수에 전달할 수 있는 종료 상태를 반환합니다.

`Py_RunMain()`을 사용하여 항상 격리 모드에서 실행되는 사용자 정의 파이썬의 예는 파이썬 구성을 참조하십시오.

10.11 다단계 초기화 비공개 잠정적 API

이 섹션은 **PEP 432**의 핵심 기능인 다단계 초기화를 소개하는 비공개 잠정적 API입니다:

- “핵심(Core)” 초기화 단계, “최소한의 파이썬”:
 - 내장형;
 - 내장 예외;
 - 내장과 프로즌 모듈(frozen modules);
 - sys 모듈은 부분적으로만 초기화됩니다(예를 들어: `sys.path`는 아직 존재하지 않습니다).
- “주(Main)” 초기화 단계, 파이썬이 완전히 초기화됩니다:
 - `importlib`를 설치하고 구성합니다;
 - 경로 구성을 적용합니다;
 - 시그널 처리기를 설치합니다;
 - sys 모듈 초기화를 완료합니다(예를 들어: `sys.stdout`과 `sys.path`를 만듭니다);
 - `faulthandler`와 `tracemalloc`과 같은 선택적 기능을 활성화합니다;
 - `site` 모듈을 임포트 합니다;
 - 등등

비공개 잠정적 API:

- `PyConfig._init_main`: 0으로 설정되면, `Py_InitializeFromConfig()`는 “핵심” 초기화 단계에서 중단합니다.

`PyStatus_Py_InitializeMain(void)`

“주” 초기화 단계로 이동하여, 파이썬 초기화를 완료합니다.

“핵심” 단계에서는 아무런 모듈도 임포트 하지 않고 `importlib` 모듈이 구성되지 않습니다: 경로 구성은 “주” 단계에서만 적용됩니다. 경로 구성을 재정의하거나 조정하기 위해 파이썬에서 파이썬을 사용자 정의할 수 있으며, 사용자 정의 `sys.meta_path` 임porter(impoter)나 임포트 혹은 등을 설치할 수 있습니다.

핵심 단계 이후에 주 단계 이전에 파이썬에서 경로 구성을 계산할 수 있게 될 수 있고, 이것이 **PEP 432**의 동기 중 하나입니다.

“핵심” 단계가 제대로 정의되지 않았습니: 이 단계에서 무엇을 사용할 수 있고, 무엇이 그렇지 않아야 하는지는 아직 지정되지 않았습니. API는 비공개이자 잠정적인 것으로 표시됩니다: 적절한 공개 API가 설계될 때까지 언제든지 API를 수정하거나 제거할 수 있습니다.

“핵심”과 “주” 초기화 단계 사이에서 파이썬 코드를 실행하는 예제:

```
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)) {
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    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);
}
}
```

11.1 개요

파이썬의 메모리 관리에는 모든 파이썬 객체와 데이터 구조를 포함하는 비공개 힙(private heap)을 수반합니다. 이 비공개 힙의 관리는 파이썬 메모리 관리자에 의해 내부적으로 이루어집니다. 파이썬 메모리 관리자는 공유, 세그먼트화, 사전 할당 또는 캐싱과 같은 동적 스토리지 관리의 다양한 측면을 처리하는 서로 다른 구성 요소를 가지고 있습니다.

가장 낮은 수준에서, 원시 메모리 할당자는 운영 체제의 메모리 관리자와 상호 작용하여 비공개 힙에 모든 파이썬 관련 데이터를 저장하기에 충분한 공간이 있는지 확인합니다. 원시 메모리 할당자 위에, 여러 개의 객체별 할당자가 같은 힙에서 작동하며 각 객체 형의 특성에 맞는 고유한 메모리 관리 정책을 구현합니다. 예를 들어, 정수는 다른 스토리지 요구 사항과 속도/공간 절충을 의미하므로, 정수 객체는 힙 내에서 문자열, 튜플 또는 딕셔너리와는 다르게 관리됩니다. 따라서 파이썬 메모리 관리자는 일부 작업을 객체별 할당자에게 위임하지만, 후자가 비공개 힙의 경계 내에서 작동하도록 합니다.

파이썬 힙의 관리는 인터프리터 자체에 의해 수행되며, 사용자는 힙 내부의 메모리 블록에 대한 객체 포인터를 규칙적으로 조작하더라도, 사용자가 제어할 수 없다는 것을 이해하는 것이 중요합니다. 파이썬 객체와 기타 내부 버퍼를 위한 힙 공간 할당은 이 설명서에 나열된 파이썬/C API 함수를 통해 파이썬 메모리 관리자의 요청에 따라 수행됩니다.

메모리 손상을 피하고자, 확장 작성자는 C 라이브러리에서 내보낸 함수를 파이썬 객체에 대해 실행하지 않아야 합니다: `malloc()`, `calloc()`, `realloc()` 및 `free()`. 그렇게 한다면, 서로 다른 알고리즘을 구현하고 다른 힙에 작동하기 때문에, C 할당자와 파이썬 메모리 관리자 간에 혼잡 호출이 발생하여 치명적인 결과를 초래합니다. 그러나, 다음 예제와 같이 개별 목적으로 C 라이브러리 할당자를 사용하여 메모리 블록을 안전하게 할당하고 해제할 수 있습니다:

```
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);
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
free(buf); /* malloc'ed */
return res;
```

이 예에서, I/O 버퍼에 대한 메모리 요청은 C 라이브러리 할당자에 의해 처리됩니다. 파이썬 메모리 관리자는 결과로 반환되는 바이트열 객체의 할당에만 관여합니다.

그러나 대부분의 경우, 파이썬 힙에서 메모리를 할당하는 것이 좋습니다. 파이썬 힙은 파이썬 메모리 관리자가 제어하기 때문입니다. 예를 들어, 인터프리터가 C로 작성된 새로운 객체 형으로 확장될 때 필요합니다. 파이썬 힙을 사용하는 또 다른 이유는 확장 모듈의 메모리 요구에 대해 파이썬 메모리 관리자에게 알리고자 하는 것입니다. 요청된 메모리가 내부적이고 매우 특정한 목적으로만 사용될 때도, 모든 메모리 요청을 파이썬 메모리 관리자에 위임하면 인터프리터가 전체 메모리 요구량에 대한 더 정확한 이미지를 갖게 됩니다. 결과적으로, 특정 상황에서, 파이썬 메모리 관리자는 가비지 수집, 메모리 압축 또는 기타 예방 절차와 같은 적절한 작업을 트리거 하거나 그러지 않을 수 있습니다. 앞의 예에서와 같이 C 라이브러리 할당자를 사용하면, I/O 버퍼에 할당된 메모리가 파이썬 메모리 관리자를 완전히 우회하게 됨에 유의하십시오.

더 보기:

PYTHONMALLOC 환경 변수를 사용하여 파이썬에서 사용하는 메모리 할당자를 구성할 수 있습니다.

PYTHONMALLOCSTATS 환경 변수는 새로운 `pymalloc` 객체 아레나(arena)가 만들어질 때마다 그리고 종료 시 `pymalloc` 메모리 할당자의 통계를 인쇄하는 데 사용될 수 있습니다.

11.2 원시 메모리 인터페이스

다음 함수 집합은 시스템 할당자에 대한 래퍼입니다. 이러한 함수는 스레드 안전해서, *GIL*을 유지할 필요는 없습니다.

기본 원시 메모리 할당자는 다음 함수를 사용합니다: `malloc()`, `calloc()`, `realloc()` 및 `free()`; 0바이트를 요청할 때 `malloc(1)`(또는 `calloc(1, 1)`)을 호출합니다.

버전 3.4에 추가.

void* PyMem_RawMalloc (size_t n)

n 바이트를 할당하고 할당된 메모리를 가리키는 void* 형의 포인터를 반환하거나, 요청이 실패하면 NULL을 반환합니다.

0바이트를 요청하면 가능하면 `PyMem_RawMalloc(1)` 이 대신 호출된 것처럼 가능하면 고유한 NULL이 아닌 포인터를 반환합니다. 메모리는 어떤 식으로든 초기화되지 않습니다.

void* PyMem_RawCalloc (size_t nelem, size_t elsize)

크기가 각각 *elsize* 바이트인 *nelem* 개의 요소를 할당하고 할당된 메모리를 가리키는 void* 형의 포인터를 반환하거나, 요청이 실패하면 NULL을 반환합니다. 메모리는 0으로 초기화됩니다.

0개의 요소나 0바이트 크기의 요소를 요청하면 `PyMem_RawCalloc(1, 1)` 이 대신 호출된 것처럼 가능하면 고유한 NULL이 아닌 포인터를 반환합니다.

버전 3.5에 추가.

void* PyMem_RawRealloc (void *p, size_t n)

*p*가 가리키는 메모리 블록의 크기를 *n* 바이트로 조정합니다. 내용은 이전과 새로운 크기의 최솟값 내에서는 변경되지 않습니다.

*p*가 NULL이면, 호출은 `PyMem_RawMalloc(n)` 과 동등합니다; *n*이 0과 같으면, 메모리 블록의 크기는 조정되지만 해제되지는 않고, 반환된 포인터는 NULL이 아닙니다.

*p*가 NULL이 아닌 한, `PyMem_RawMalloc()`, `PyMem_RawRealloc()` 또는 `PyMem_RawCalloc()`에 대한 이전 호출에 의해 반환된 것이어야 합니다.

요청이 실패하면, `PyMem_RawRealloc()` 은 NULL을 반환하고 `p`는 이전 메모리 영역에 대한 유효한 포인터로 유지됩니다.

void **PyMem_RawFree** (void **p*)

`p`가 가리키는 메모리 블록을 해제합니다. `p`는 `PyMem_RawMalloc()`, `PyMem_RawRealloc()` 또는 `PyMem_RawCalloc()`에 대한 이전 호출로 반환된 것이어야 합니다. 그렇지 않거나 `PyMem_RawFree(p)`가 앞서 호출되었으면, 정의되지 않은 동작이 일어납니다.

`p`가 NULL이면, 아무 작업도 수행되지 않습니다.

11.3 메모리 인터페이스

ANSI C 표준에 따라 모델링 되었지만 0바이트를 요청할 때의 동작을 지정한 다음 함수 집합은 파이썬 힙에서 메모리를 할당하고 해제하는 데 사용할 수 있습니다.

기본 메모리 할당자는 `pymalloc` 메모리 할당자를 사용합니다.

경고: 이 함수를 사용할 때는 `GIL`을 유지해야 합니다.

버전 3.6에서 변경: 기본 할당자는 이제 시스템 `malloc()` 대신 `pymalloc` 입니다.

void* **PyMem_Malloc** (size_t *n*)

`n` 바이트를 할당하고 할당된 메모리를 가리키는 void* 형의 포인터를 반환하거나, 요청이 실패하면 NULL을 반환합니다.

0바이트를 요청하면 `PyMem_Malloc(1)` 이 대신 호출된 것처럼 가능하면 고유한 NULL이 아닌 포인터를 반환합니다. 메모리는 어떤 식으로든 초기화되지 않습니다.

void* **PyMem_Calloc** (size_t *nelem*, size_t *elsize*)

크기가 각각 `elsize` 바이트인 `nelem` 개의 요소를 할당하고 할당된 메모리를 가리키는 void* 형의 포인터를 반환하거나, 요청이 실패하면 NULL을 반환합니다. 메모리는 0으로 초기화됩니다.

0개의 요소나 0바이트 크기의 요소를 요청하면 `PyMem_Calloc(1, 1)` 이 대신 호출된 것처럼 가능하면 고유한 NULL이 아닌 포인터를 반환합니다.

버전 3.5에 추가.

void* **PyMem_Realloc** (void **p*, size_t *n*)

`p`가 가리키는 메모리 블록의 크기를 `n` 바이트로 조정합니다. 내용은 이전과 새로운 크기의 최솟값 내에 서는 변경되지 않습니다.

`p`가 NULL이면, 호출은 `PyMem_Malloc(n)` 과 동등합니다; 그렇지 않고 `n`이 0과 같으면, 메모리 블록의 크기는 조정되지만 해제되지는 않으며, 반환된 포인터는 NULL이 아닙니다.

`p`가 NULL이 아닌 한, `PyMem_Malloc()`, `PyMem_Realloc()` 또는 `PyMem_Calloc()`에 대한 이전 호출이 반환한 것이어야 합니다.

요청이 실패하면, `PyMem_Realloc()` 은 NULL을 반환하고 `p`는 이전 메모리 영역에 대한 유효한 포인터로 유지됩니다.

void **PyMem_Free** (void **p*)

`p`가 가리키는 메모리 블록을 해제합니다. `p`는 `PyMem_Malloc()`, `PyMem_Realloc()` 또는 `PyMem_Calloc()`에 대한 이전 호출이 반환한 것이어야 합니다. 그렇지 않거나 `PyMem_Free(p)`가 앞서 호출되었으면 정의되지 않은 동작이 일어납니다.

`p`가 NULL이면, 아무 작업도 수행되지 않습니다.

편의를 위해 다음과 같은 형 지향 매크로가 제공됩니다. `TYPE`이 모든 C형을 나타냄에 유의하십시오.

TYPE* **PyMem_New** (TYPE, size_t *n*)

PyMem_Malloc() 과 같지만, ($n * \text{sizeof}(\text{TYPE})$) 바이트의 메모리를 할당합니다. TYPE*로 캐스트된 포인터를 반환합니다. 메모리는 어떤 식으로든 초기화되지 않습니다.

TYPE* **PyMem_Resize** (void **p*, TYPE, size_t *n*)

PyMem_Realloc() 과 같지만, 메모리 블록의 크기는 ($n * \text{sizeof}(\text{TYPE})$) 바이트로 조정됩니다. TYPE*로 캐스트된 포인터를 반환합니다. 반환한 후에, *p*는 새로운 메모리 영역에 대한 포인터이거나, 실패하면 NULL이 됩니다.

이것은 C 전처리기 매크로입니다; *p*는 항상 다시 대입됩니다. 에러를 처리할 때 메모리 손실을 피하려면 *p*의 원래 값을 보관하십시오.

void **PyMem_Del** (void **p*)

PyMem_Free() 와 같습니다.

또한, 위에 나열된 C API 함수를 사용하지 않고, 파이썬 메모리 할당자를 직접 호출하기 위해 다음 매크로 집합이 제공됩니다. 그러나, 이들을 사용하면 파이썬 버전을 가로지르는 바이너리 호환성이 유지되지 않아서 확장 모듈에서는 폐지되었습니다.

- PyMem_MALLOC(size)
- PyMem_NEW(type, size)
- PyMem_REALLOC(ptr, size)
- PyMem_RESIZE(ptr, type, size)
- PyMem_FREE(ptr)
- PyMem_DEL(ptr)

11.4 객체 할당자

ANSI C 표준에 따라 모델링 되었지만 0바이트를 요청할 때의 동작을 지정한 다음 함수 집합은 파이썬 힙에서 메모리를 할당하고 해제하는 데 사용할 수 있습니다.

기본 객체 할당자는 *pymalloc* 메모리 할당자를 사용합니다.

경고: 이 함수를 사용할 때는 *GIL*을 유지해야 합니다.

void* **PyObject_Malloc** (size_t *n*)

n 바이트를 할당하고 할당된 메모리를 가리키는 void* 형의 포인터를 반환하거나, 요청이 실패하면 NULL을 반환합니다.

0바이트를 요청하면 PyObject_Malloc(1) 이 대신 호출된 것처럼 가능한 고유한 NULL이 아닌 포인터를 반환합니다. 메모리는 어떤 식으로든 초기화되지 않습니다.

void* **PyObject_Calloc** (size_t *nelem*, size_t *elsize*)

크기가 각각 *elsize* 바이트인 *nelem* 개의 요소를 할당하고 할당된 메모리를 가리키는 void* 형의 포인터를 반환하거나, 요청이 실패하면 NULL을 반환합니다. 메모리는 0으로 초기화됩니다.

0개의 요소나 0바이트 크기의 요소를 요청하면 PyObject_Calloc(1, 1) 이 대신 호출된 것처럼 가능한 고유한 NULL이 아닌 포인터를 반환합니다.

버전 3.5에 추가.

void* **PyObject_Realloc** (void **p*, size_t *n*)

*p*가 가리키는 메모리 블록의 크기를 *n* 바이트로 조정합니다. 내용은 이전과 새로운 크기의 최솟값 내에서는 변경되지 않습니다.

p 가 NULL이면, 호출은 `PyObject_Malloc(n)` 과 동등합니다; 그렇지 않고 n 이 0과 같으면, 메모리 블록의 크기는 조정되지만 해제되지 않고, 반환된 포인터는 NULL이 아닙니다.

p 가 NULL이 아닌 한, `PyObject_Malloc()`, `PyObject_Realloc()` 또는 `PyObject_Calloc()`에 대한 이전 호출에 의해 반환된 것이어야 합니다.

요청이 실패하면, `PyObject_Realloc()`은 NULL을 반환하고 p 는 이전 메모리 영역에 대한 유효한 포인터로 유지됩니다.

void **PyObject_Free** (void * p)

p 가 가리키는 메모리 블록을 해제합니다. 이 블록은 `PyObject_Malloc()`, `PyObject_Realloc()` 또는 `PyObject_Calloc()`에 대한 이전 호출에 의해 반환된 것이어야 합니다. 그렇지 않거나 `PyObject_Free(p)`가 이전에 호출되었으면 정의되지 않은 동작이 일어납니다.

p 가 NULL이면, 아무 작업도 수행되지 않습니다.

11.5 기본 메모리 할당자

기본 메모리 할당자:

구성	이름	PyMem_RawMalloc	PyMem_Malloc	PyObject_Malloc
릴리스 빌드	"pymalloc"	malloc	pymalloc	pymalloc
디버그 빌드	"pymalloc_debug"	malloc + 디버그	pymalloc + 디버그	pymalloc + 디버그
pymalloc 없는 배포 빌드	"malloc"	malloc	malloc	malloc
pymalloc 없는 디버그 빌드	"malloc_debug"	malloc + 디버그	malloc + 디버그	malloc + 디버그

범례:

- 이름: PYTHONMALLOC 환경 변수의 값
- malloc: 표준 C 라이브러리의 시스템 할당자, C 함수: `malloc()`, `calloc()`, `realloc()` 및 `free()`
- pymalloc: *pymalloc* 메모리 할당자
- "+ 디버그": `PyMem_SetupDebugHooks()`에 의해 설치된 디버그 훅 포함

11.6 메모리 할당자 사용자 정의

버전 3.4에 추가.

PyMemAllocatorEx

메모리 블록 할당자를 기술하는 데 사용되는 구조체. 구조체에는 네 개의 필드가 있습니다:

필드	의미
<code>void *ctx</code>	첫 번째 인자로 전달된 사용자 컨텍스트
<code>void* malloc(void *ctx, size_t size)</code>	메모리 블록을 할당합니다
<code>void* calloc(void *ctx, size_t nelem, size_t elsize)</code>	0으로 초기화된 메모리 블록을 할당합니다
<code>void* realloc(void *ctx, void *ptr, size_t new_size)</code>	메모리 블록을 할당하거나 크기 조정합니다
<code>void free(void *ctx, void *ptr)</code>	메모리 블록을 해제합니다

버전 3.5에서 변경: `PyMemAllocator` 구조체의 이름이 `PyMemAllocatorEx`로 바뀌고 새로운 `calloc` 필드가 추가되었습니다.

PyMemAllocatorDomain

할당자 도메인을 식별하는 데 사용되는 열거형. 도메인:

PYMEM_DOMAIN_RAW

함수:

- `PyMem_RawMalloc()`
- `PyMem_RawRealloc()`
- `PyMem_RawCalloc()`
- `PyMem_RawFree()`

PYMEM_DOMAIN_MEM

함수:

- `PyMem_Malloc()`,
- `PyMem_Realloc()`
- `PyMem_Calloc()`
- `PyMem_Free()`

PYMEM_DOMAIN_OBJ

함수:

- `PyObject_Malloc()`
- `PyObject_Realloc()`
- `PyObject_Calloc()`
- `PyObject_Free()`

void **PyMem_GetAllocator** (*PyMemAllocatorDomain domain, PyMemAllocatorEx *allocator*)

지정된 도메인의 메모리 블록 할당자를 가져옵니다.

void **PyMem_SetAllocator** (*PyMemAllocatorDomain domain, PyMemAllocatorEx *allocator*)

지정된 도메인의 메모리 블록 할당자를 설정합니다.

새 할당자는 0바이트를 요청할 때 고유한 NULL이 아닌 포인터를 반환해야 합니다.

`PYMEM_DOMAIN_RAW` 도메인의 경우, 할당자는 스레드 안전해야 합니다: 할당자가 호출될 때 *GIL*을 잡지 않습니다.

새 할당자가 혹이 아니면 (이전 할당자를 호출하지 않으면), `PyMem_SetupDebugHooks()` 함수를 호출하여 새 할당자 위에 디버그 혹을 다시 설치해야 합니다.

void **PyMem_SetupDebugHooks** (void)

파이썬 메모리 할당자 함수에 있는 버그를 감지하기 위한 훅을 설정합니다.

새로 할당된 메모리는 바이트 0xCD(CLEANBYTE)로 채워지고, 해제된 메모리는 바이트 0xDD(DEADBYTE)로 채워집니다. 메모리 블록은 “금지된 바이트”(FORBIDDENBYTE: 바이트 0xFD)로 둘러싸여 있습니다.

실행 시간 검사:

- API 위반 탐지, 예: `PyMem_Malloc()` 이 할당한 버퍼에 대해 호출된 `PyObject_Free()`
- 버퍼 시작 전에 쓰기 감지 (버퍼 언더플로)
- 버퍼 끝 뒤에 쓰기 감지 (버퍼 오버플로)
- `PYMEM_DOMAIN_OBJ`(예 : `PyObject_Malloc()`) 와 `PYMEM_DOMAIN_MEM`(예 : `PyMem_Malloc()`) 도메인의 할당자 함수가 호출될 때 `GIL`이 유지되는지 확인

에러가 발생하면, 디버그 훅은 `tracemalloc` 모듈을 사용하여 메모리 블록이 할당된 곳의 트레이스백을 가져옵니다. `tracemalloc`이 파이썬 메모리 할당을 추적 중이고 메모리 블록이 추적될 때만 트레이스백이 표시됩니다.

파이썬이 디버그 모드에서 컴파일되면 이러한 훅은 기본적으로 설치됩니다. `PYTHONMALLOC` 환경 변수를 사용하여 릴리스 모드에서 컴파일된 파이썬에 디버그 훅을 설치할 수 있습니다.

버전 3.6에서 변경: 이 함수는 이제 릴리스 모드에서 컴파일된 파이썬에서도 작동합니다. 에러가 발생하면, 디버그 훅은 이제 `tracemalloc`을 사용하여 메모리 블록이 할당된 곳의 트레이스백을 가져옵니다. 또한 디버그 훅은 이제 `PYMEM_DOMAIN_OBJ`와 `PYMEM_DOMAIN_MEM` 도메인의 함수가 호출될 때 `GIL`을 잡는지 확인합니다.

버전 3.8에서 변경: 바이트 패턴 0xCB(CLEANBYTE), 0xDB(DEADBYTE) 및 0xFB(FORBIDDENBYTE)는 윈도우 CRT 디버그 `malloc()` 및 `free()`와 같은 값을 사용하도록 0xCD, 0xDD 및 0xFD로 대체되었습니다.

11.7 pymalloc 할당자

파이썬에는 수명이 짧은 작은 (512바이트 이하) 객체에 최적화된 `pymalloc` 할당자가 있습니다. 256 KiB의 고정 크기를 갖는 “아레나(arena)”라는 메모리 매핑을 사용합니다. 512 바이트보다 큰 할당의 경우 `PyMem_RawMalloc()`과 `PyMem_RawRealloc()`으로 대체됩니다.

`pymalloc` 은 `PYMEM_DOMAIN_MEM`(예 : `PyMem_Malloc()`) 과 `PYMEM_DOMAIN_OBJ`(예 : `PyObject_Malloc()`) 도메인의 기본 할당자입니다.

아레나 할당자는 다음 함수를 사용합니다:

- 윈도우에서 `VirtualAlloc()`과 `VirtualFree()`,
- 사용할 수 있으면 `mmap()`과 `munmap()`
- 그렇지 않으면 `malloc()`과 `free()`

11.7.1 pymalloc 아레나 할당자 사용자 정의

버전 3.4에 추가.

PyObjectArenaAllocator

아레나 할당자를 기술하는 데 사용되는 구조체. 이 구조체에는 세 개의 필드가 있습니다:

필드	의미
void *ctx	첫 번째 인자로 전달된 사용자 컨텍스트
void* alloc(void *ctx, size_t size)	size 바이트의 아레나를 할당합니다
void free(void *ctx, void *ptr, size_t size)	아레나를 해제합니다

void **PyObject_GetArenaAllocator** (*PyObjectArenaAllocator* *allocator)

아레나 할당자를 얻습니다.

void **PyObject_SetArenaAllocator** (*PyObjectArenaAllocator* *allocator)

아레나 할당자를 설정합니다.

11.8 tracemalloc C API

버전 3.7에 추가.

int **PyTraceMalloc_Track** (unsigned int domain, uintptr_t ptr, size_t size)

tracemalloc 모듈에서 할당된 메모리 블록을 추적합니다.

성공하면 0을 반환하고, 에러가 발생하면 (추적을 저장하기 위한 메모리를 할당하지 못했습니다) -1을 반환합니다. tracemalloc이 비활성화되었으면 -2를 반환합니다.

메모리 블록이 이미 추적되면, 기존 추적을 갱신합니다.

int **PyTraceMalloc_Untrack** (unsigned int domain, uintptr_t ptr)

tracemalloc 모듈에서 할당된 메모리 블록을 추적 해제합니다. 블록이 추적되지 않으면 아무것도 하지 않습니다.

tracemalloc이 비활성화되었으면 -2를 반환하고, 그렇지 않으면 0을 반환합니다.

11.9 예

다음은 개요 섹션에서 따온 예제입니다. I/O 버퍼가 첫 번째 함수 집합을 사용하여 파이썬 힙에서 할당되도록 다시 작성되었습니다:

```
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;
```

형 지향 함수 집합을 사용하는 같은 코드입니다:

```
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;
```

위의 두 가지 예에서, 버퍼는 항상 같은 집합에 속하는 함수를 통해 조작됨에 유의하십시오. 실제로, 서로 다른 할당자를 혼합할 위험이 최소로 줄어들도록, 주어진 메모리 블록에 대해 같은 메모리 API 패밀리를 사용하는 것은 필수입니다. 다음 코드 시퀀스에는 두 개의 예러가 있으며, 그중 하나는 서로 다른 힙에서 작동하는 두 개의 다른 할당자를 혼합하기 때문에 치명적(*fatal*)인 것으로 표시됩니다.

```
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() */
```

파이썬 힙에서 원시 메모리 블록을 처리하기 위한 함수 외에도, 파이썬의 객체는 `PyObject_New()`, `PyObject_NewVar()` 및 `PyObject_Del()`로 할당되고 해제됩니다.

이것들은 C로 새로운 객체 형을 정의하고 구현하는 것에 대한 다음 장에서 설명될 것입니다.

이 장에서는 새 객체 형을 정의할 때 사용되는 함수, 형 및 매크로에 관해 설명합니다.

12.1 힙에 객체 할당하기

*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. 새로 할당된 객체 *op*를 형과 초기 참조로 초기화합니다. 초기화된 객체를 반환합니다. *type*이 객체가 순환 가비지 감지기에 참여함을 나타내면, 감지기의 감시되는 객체 집합에 추가됩니다. 객체의 다른 필드는 영향을 받지 않습니다.

*PyVarObject** **PyObject_InitVar** (*PyVarObject* *op, *PyTypeObject* *type, *Py_ssize_t* size)

Return value: Borrowed reference. 이것은 *PyObject_Init()*가 수행하는 모든 작업을 수행하고, 가변 크기 객체의 길이 정보도 초기화합니다.

*TYPE** **PyObject_New** (*TYPE*, *PyTypeObject* *type)

Return value: New reference. C 구조체 형 *TYPE*과 파이썬 형 객체 *type*을 사용하여 새로운 파이썬 객체를 할당합니다. 파이썬 객체 헤더로 정의되지 않은 필드는 초기화되지 않습니다; 객체의 참조 횟수는 1이 됩니다. 메모리 할당의 크기는 형 객체의 *tp_basicsize* 필드에서 결정됩니다.

*TYPE** **PyObject_NewVar** (*TYPE*, *PyTypeObject* *type, *Py_ssize_t* size)

Return value: New reference. C 구조체 형 *TYPE*과 파이썬 타입 형 *type*을 사용하여 새로운 파이썬 객체를 할당합니다. 파이썬 객체 헤더로 정의되지 않은 필드는 초기화되지 않습니다. 할당된 메모리는 *TYPE* 구조체에 더해 *type*의 *tp_itemsize* 필드에 의해 주어진 크기의 *size* 필드를 허용합니다. 이는 튜플과 같은 객체를 구현할 때 유용합니다. 튜플은 만들 때 크기를 결정할 수 있습니다. 같은 할당에 필드 배열을 포함 시키면, 할당 횟수가 줄어들어, 메모리 관리 효율성이 향상됩니다.

void PyObject_Del (*void* *op)

PyObject_New() 나 *PyObject_NewVar()*를 사용한 객체에 할당된 메모리를 해제합니다. 이것은

일반적으로 객체의 형에 지정된 `tp_dealloc` 처리기에서 호출됩니다. 메모리가 더는 유효한 파이썬 객체가 아니므로, 이 호출 후에는 객체의 필드에 액세스해서는 안 됩니다.

`PyObject_Py_NoneStruct`

파이썬에서 `None`으로 노출되는 객체. 이 객체에 대한 포인터로 평가되는 `Py_None` 매크로를 사용해서 액세스해야 합니다.

더 보기:

`PyModule_Create()` 확장 모듈을 할당하고 만듭니다.

12.2 공통 객체 구조체

파이썬의 객체 형 정의에 사용되는 많은 구조체가 있습니다. 이 섹션에서는 이러한 구조체와 사용 방법에 대해 설명합니다.

모든 파이썬 객체는 궁극적으로 객체의 메모리 표현의 처음에서 적은 수의 필드를 공유합니다. 이들은 `PyObject`와 `PyVarObject` 형으로 표시되며, 다른 모든 파이썬 객체의 정의에서, 직접 또는 간접적으로, 사용되는 일부 매크로의 확장을 통해 정의됩니다.

`PyObject`

모든 객체 형은 이 형의 확장입니다. 이것은 파이썬이 객체에 대한 포인터를 객체로 취급하는 데 필요한 정보를 포함하는 형입니다. 일반적인 “릴리스” 빌드에는, 객체의 참조 횟수와 해당 형 객체에 대한 포인터만 포함됩니다. 실제로 `PyObject`로 선언된 것은 없지만, 파이썬 객체에 대한 모든 포인터를 `PyObject*`로 캐스트 할 수 있습니다. `Py_REFCNT`와 `Py_TYPE` 매크로를 사용하여 멤버에 액세스해야 합니다.

`PyVarObject`

이것은 `ob_size` 필드를 추가하는 `PyObject`의 확장입니다. 이것은 길이라는 개념을 가진 객체에만 사용됩니다. 이 형은 종종 파이썬/C API에 나타나지 않습니다. `Py_REFCNT`, `Py_TYPE` 및 `Py_SIZE` 매크로를 사용하여 멤버에 액세스해야 합니다.

`PyObject_HEAD`

길이가 변하지 않는 객체를 나타내는 새로운 형을 선언할 때 사용되는 매크로입니다. `PyObject_HEAD` 매크로는 다음과 같이 확장됩니다:

```
PyObject ob_base;
```

위의 `PyObject` 설명서를 참조하십시오.

`PyObject_VAR_HEAD`

인스턴스마다 길이가 다른 객체를 나타내는 새로운 형을 선언할 때 사용되는 매크로입니다. `PyObject_VAR_HEAD` 매크로는 다음과 같이 확장됩니다:

```
PyVarObject ob_base;
```

위의 `PyVarObject` 설명서를 참조하십시오.

`Py_TYPE(o)`

이 매크로는 파이썬 객체의 `ob_type` 멤버에 액세스하는 데 사용됩니다. 다음으로 확장됩니다:

```
((PyObject*) (o))->ob_type)
```

`Py_REFCNT(o)`

이 매크로는 파이썬 객체의 `ob_refcnt` 멤버에 액세스하는 데 사용됩니다. 다음으로 확장됩니다:

```
((PyObject*) (o))->ob_refcnt)
```

Py_SIZE(o)

이 매크로는 파이썬 객체의 `ob_size` 멤버에 액세스하는 데 사용됩니다. 다음으로 확장됩니다:

```
((PyVarObject*)(o))->ob_size)
```

PyObject_HEAD_INIT(type)

이것은 새로운 *PyObject* 형의 초기화 값으로 확장되는 매크로입니다. 이 매크로는 다음으로 확장됩니다:

```
_PyObject_EXTRA_INIT  
1, type,
```

PyVarObject_HEAD_INIT(type, size)

이것은 `ob_size` 필드를 포함하여, 새로운 *PyVarObject* 형의 초기화 값으로 확장되는 매크로입니다. 이 매크로는 다음으로 확장됩니다:

```
_PyObject_EXTRA_INIT  
1, type, size,
```

PyCFunction

대부분 파이썬 콜러블을 C로 구현하는 데 사용되는 함수 형. 이 형의 함수는 두 개의 *PyObject** 매개 변수를 취하고 하나의 값을 반환합니다. 반환 값이 *NULL*이면, 예외가 설정되어 있어야 합니다. *NULL*이 아니면, 반환 값은 파이썬에 노출된 함수의 반환 값으로 해석됩니다. 함수는 새로운 참조를 반환해야 합니다.

PyCFunctionWithKeywords

서명이 `METH_VARARGS | METH_KEYWORDS`인 파이썬 콜러블을 C로 구현하는 데 사용되는 함수 형.

_PyCFunctionFast

서명이 `METH_FASTCALL`인 파이썬 콜러블을 C로 구현하는 데 사용되는 함수 형.

_PyCFunctionFastWithKeywords

서명이 `METH_FASTCALL | METH_KEYWORDS`인 파이썬 콜러블을 C로 구현하는 데 사용되는 함수 형.

PyMethodDef

확장형의 메서드를 기술하는 데 사용되는 구조체. 이 구조체에는 네 개의 필드가 있습니다:

필드	C 형	의미
<code>ml_name</code>	<code>const char *</code>	메서드의 이름
<code>ml_meth</code>	<code>PyCFunction</code>	C 구현에 대한 포인터
<code>ml_flags</code>	<code>int</code>	호출 구성 방법을 나타내는 플래그 비트
<code>ml_doc</code>	<code>const char *</code>	독스트링의 내용을 가리킵니다

`ml_meth`는 C 함수 포인터입니다. 함수는 형이 다를 수 있지만, 항상 *PyObject**를 반환합니다. 함수가 *PyCFunction*이 아니면, 컴파일러는 메서드 테이블에서 캐스트를 요구합니다. *PyCFunction*이 첫 번째 매개 변수를 *PyObject**로 정의하더라도, 일반적으로 메서드 구현은 *self* 객체의 특정 C 형을 사용합니다.

`ml_flags` 필드는 다음 플래그를 포함 할 수 있는 비트 필드입니다. 개별 플래그는 호출 규칙이나 바인딩 규칙을 나타냅니다.

위치 인자에 대한 네 가지 기본 호출 규칙이 있으며 이 중 두 가지는 `METH_KEYWORDS`와 결합하여 키워드 인자도 지원할 수 있습니다. 그래서 총 6개의 호출 규칙이 있습니다:

METH_VARARGS

이는 메서드가 *PyCFunction* 형인 일반적인 호출 규칙입니다. 함수는 두 개의 *PyObject** 값을 기대합니다. 첫 번째는 메서드의 *self* 객체입니다; 모듈 함수의 경우, 모듈 객체입니다. 두 번째 매개 변수(종종 *args*라고 합니다)는 모든 인자를 나타내는 튜플 객체입니다. 이 매개 변수는 일반적으로 *PyArg_ParseTuple()* 이나 *PyArg_UnpackTuple()* 을 사용하여 처리됩니다.

METH_VARARGS | METH_KEYWORDS

이러한 플래그가 있는 메서드는 `PyCFunctionWithKeywords` 형이어야 합니다. 이 함수는 세 개의 매개 변수를 기대합니다: `self`, `args`, `kwargs`. 여기서 `kwargs`는 모든 키워드 인자의 딕셔너리이거나 키워드 인자가 없으면 `NULL`일 수 있습니다. 매개 변수는 일반적으로 `PyArg_ParseTupleAndKeywords()`를 사용하여 처리됩니다.

METH_FASTCALL

위치 인자만 지원하는 빠른 호출 규칙. 메서드의 형은 `_PyCFunctionFast`입니다. 첫 번째 매개 변수는 `self`이고, 두 번째 매개 변수는 인자를 나타내는 `PyObject*` 값의 C 배열이며, 세 번째 매개 변수는 인자 수(배열의 길이)입니다.

이것은 제한된 *API*의 일부가 아닙니다.

버전 3.7에 추가.

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 vector-call protocol: there is an additional fourth `PyObject*` parameter which is a tuple representing the names of the keyword arguments or possibly `NULL` if there are no keywords. The values of the keyword arguments are stored in the `args` array, after the positional arguments.

이것은 제한된 *API*의 일부가 아닙니다.

버전 3.7에 추가.

METH_NOARGS

매개 변수가 없는 메서드는 `METH_NOARGS` 플래그로 나열되어 있으면, 인자가 주어졌는지 확인할 필요가 없습니다. `PyCFunction` 형이어야 합니다. 첫 번째 매개 변수의 이름은 일반적으로 `self`이며 모듈이나 객체 인스턴스에 대한 참조를 보유합니다. 모든 경우에 두 번째 매개 변수는 `NULL`입니다.

METH_O

"O" 인자로 `PyArg_ParseTuple()`을 호출하는 대신, 단일 객체 인자가 있는 메서드는 `METH_O` 플래그로 나열할 수 있습니다. `PyCFunction` 형이고, `self` 매개 변수와 단일 인자를 나타내는 `PyObject*` 매개 변수를 갖습니다.

이 두 상수는 호출 규칙을 나타내는 데 사용되지 않고 클래스의 메서드와 함께 사용할 때 바인딩을 나타냅니다. 모듈에 정의된 함수에는 사용할 수 없습니다. 이러한 플래그 중 최대 하나를 주어진 메서드에 대해 설정할 수 있습니다.

METH_CLASS

메서드로 형의 인스턴스가 아닌 형 객체가 첫 번째 매개 변수로 전달됩니다. `classmethod()` 내장 함수를 사용할 때 만들어지는 것과 유사한 클래스 메서드(*class methods*)를 만드는 데 사용됩니다.

METH_STATIC

메서드로 형의 인스턴스가 아닌 `NULL`이 첫 번째 매개 변수로 전달됩니다. `staticmethod()` 내장 함수를 사용할 때 만들어지는 것과 유사한 정적 메서드(*static methods*)를 만드는 데 사용됩니다.

하나의 다른 상수는 같은 메서드 이름을 가진 다른 정의 대신 메서드가 로드되는지를 제어합니다.

METH_COEXIST

기존 정의 대신 메서드가 로드됩니다. `METH_COEXIST`가 없으면, 기본값은 반복되는 정의를 건너뛰는 것입니다. 슬롯 래퍼가 메서드 테이블 전에 로드되므로, 예를 들어 `sq_contains` 슬롯의 존재는 `__contains__()` 라는 래핑된 메서드를 생성하고 같은 이름의 해당 `PyCFunction`을 로드하지 않습니다. 플래그가 정의되면, `PyCFunction`이 래퍼 객체 자리에 로드되고 슬롯과 공존합니다. 이는 `PyCFunction`에 대한 호출이 래퍼 객체 호출보다 최적화되어 있기 때문에 유용합니다.

PyMemberDef

C 구조체 멤버에 해당하는 형의 어트리뷰트를 기술하는 구조체. 필드는 다음과 같습니다:

필드	C 형	의미
name	const char *	멤버의 이름
type	int	C 구조체에 있는 멤버의 형
offset	Py_ssize_t	멤버가 형의 객체 구조체에 위치하는 바이트 단위의 오프셋
flags	int	필드가 읽기 전용인지 쓰기 가능한지를 나타내는 플래그 비트
doc	const char *	독스트링의 내용을 가리킵니다

type은 다양한 C 형에 해당하는 많은 T_ 매크로 중 하나일 수 있습니다. 멤버가 파이썬에서 액세스 될 때, 동등한 파이썬 형으로 변환됩니다.

매크로 이름	C 형
T_SHORT	short
T_INT	int
T_LONG	long
T_FLOAT	float
T_DOUBLE	double
T_STRING	const char *
T_OBJECT	PyObject *
T_OBJECT_EX	PyObject *
T_CHAR	char
T_BYTE	char
T_UBYTE	unsigned char
T_UINT	unsigned int
T_USHORT	unsigned short
T_ULONG	unsigned long
T_BOOL	char
T_LONGLONG	long long
T_ULONGLONG	unsigned long long
T_PYSSIZET	Py_ssize_t

멤버가 NULL일 때 T_OBJECT는 None을 반환하고 T_OBJECT_EX는 AttributeError를 발생시킨다는 점에서 T_OBJECT와 T_OBJECT_EX가 다릅니다. T_OBJECT_EX가 T_OBJECT보다 해당 어트리뷰트에 대한 del 문 사용을 더 올바르게 처리하므로, T_OBJECT보다 T_OBJECT_EX를 사용하십시오.

flags는 쓰기과 읽기 액세스를 위해 0이거나, 읽기 전용 액세스를 위해 READONLY 일 수 있습니다. type에 T_STRING을 사용한다는 것은 READONLY를 뜻합니다. T_STRING 데이터는 UTF-8로 해석됩니다. T_OBJECT와 T_OBJECT_EX 멤버만 삭제될 수 있습니다. (NULL로 설정됩니다).

PyGetSetDef

형에 대한 프로퍼티 같은 액세스를 정의하는 구조체. *PyTypeObject.tp_getset* 슬롯에 대한 설명도 참조하십시오.

필드	C 형	의미
name	const char *	어트리뷰트 이름
get	getter	어트리뷰트를 얻는 C 함수
set	setter	어트리뷰트를 설정하거나 삭제하는 선택적 C 함수, 생략되면 어트리뷰트는 읽기 전용입니다
doc	const char *	선택적 독스트링
closure	void *	getter와 setter에 추가 데이터를 제공하는 선택적 함수 포인터

get 함수는 하나의 *PyObject** 매개 변수(인스턴스)와 함수 포인터(연관된 closure)를 받아들입니다:

```
typedef PyObject *(*getter)(PyObject *, void *);
```

성공하면 새 참조를 반환하고, 실패하면 설정된 예외와 함께 NULL을 반환해야 합니다.

set 함수는 두 개의 *PyObject** 매개 변수(인스턴스와 설정할 값)와 함수 포인터(연관된 closure)를 받아들입니다:

```
typedef int (*setter)(PyObject *, PyObject *, void *);
```

어트리뷰트를 삭제해야 하는 경우 두 번째 매개 변수는 NULL입니다. 성공하면 0을, 실패하면 설정된 예외와 함께 -1을 반환해야 합니다.

12.3 형 객체

아마도 파이썬 객체 시스템의 가장 중요한 구조체 중 하나는 새로운 형을 정의하는 구조체일 것입니다: *PyTypeObject* 구조체. *PyObject_**() 나 *PyType_**() 함수를 사용하여 형 객체를 처리할 수 있지만, 대부분 파이썬 응용 프로그램이 흥미를 느낄 것은 많이 제공하지 않습니다. 이 객체는 객체의 동작 방식의 기초를 이루므로, 인터프리터 자체와 새로운 형을 구현하는 확장 모듈에 매우 중요합니다.

형 객체는 대부분 표준형보다 상당히 큼니다. 크기가 큰 이유는 각 형 객체가 많은 수의 값을 저장하기 때문인데, 주로 C 함수 포인터이고 각기 형의 기능 중 작은 부분을 구현합니다. 이 섹션에서는 형 객체의 필드를 자세히 살펴봅니다. 필드는 구조체에서 나타나는 순서대로 설명됩니다.

다음의 간략 참조 외에도, 예 섹션은 *PyTypeObject*의 의미와 사용에 대한 통찰을 제공합니다.

12.3.1 간략 참조

“tp 슬롯”

PyTypeObject 슬롯 ¹	형	특수 메서드/어트리뷰트	정 보 ²				
			O	T	D	I	
<R> <i>tp_name</i>	const char *	__name__	X	X			
<i>tp_basicsize</i>	Py_ssize_t		X	X			X
<i>tp_itemsize</i>	Py_ssize_t			X			X
<i>tp_dealloc</i>	destructor		X	X			X
<i>tp_vectorcall_offset</i>	Py_ssize_t						?
(<i>tp_getattr</i>)	getattrfunc	__getattr__					G
(<i>tp_setattr</i>)	setattrfunc	__setattr__					G
<i>tp_as_async</i>	PyAsyncMethods *	서브 슬롯					%
<i>tp_repr</i>	reprfunc	__repr__	X	X			X
<i>tp_as_number</i>	PyNumberMethods *	서브 슬롯					%
<i>tp_as_sequence</i>	PySequenceMethods *	서브 슬롯					%
<i>tp_as_mapping</i>	PyMappingMethods *	서브 슬롯					%
<i>tp_hash</i>	hashfunc	__hash__	X				G
<i>tp_call</i>	ternaryfunc	__call__		X			X
<i>tp_str</i>	reprfunc	__str__	X				X
<i>tp_getattro</i>	getattrofunc	__getattr__	X	X			G

다음 페이지에 계속

표 1 - 이전 페이지에서 계속

PyTypeObject 슬롯 ¹	형	특수 메서드/어트리뷰트	정보 ²				
			O	T	D	I	
<code>tp_setattro</code>	<code>setattrofunc</code>	<code>__setattr__, __delattr__</code>	X	X			G
<code>tp_as_buffer</code>	<code>PyBufferProcs *</code>						%
<code>tp_flags</code>	<code>unsigned long</code>		X	X			?
<code>tp_doc</code>	<code>const char *</code>	<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__, __le__, __eq__, __ne__, __gt__, __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__, __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>	<code>unsigned int</code>						
<code>tp_finalize</code>	<code>destructor</code>	<code>__del__</code>					X

¹ 괄호 안의 슬롯 이름은 슬롯이 (효과적으로) 폐지되었음을 나타냅니다. 화살 괄호(angle brackets) 안에 있는 이름은 읽기 전용으로 취급해야 합니다. 대괄호(square brackets) 안의 이름은 내부 전용입니다. (점두사일 때) “<R>”는 필드가 필수임을 뜻합니다 (반드시 NULL이 아니어야 합니다).

² 열:

“O”: `PyBaseObject_Type`에 설정

“T”: `PyType_Type`에 설정

“D”: 기본값 (슬롯이 NULL로 설정된 경우)

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”: 상속

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

COUNT_ALLOCS가 정의되면 다음 (내부 전용) 필드도 존재합니다:

- `tp_allocs`
- `tp_frees`
- `tp_maxalloc`
- `tp_prev`
- `tp_next`

서브 슬롯

슬롯	형	특수 메서드
<code>am_await</code>	<code>unaryfunc</code>	<code>__await__</code>
<code>am_aiter</code>	<code>unaryfunc</code>	<code>__aiter__</code>
<code>am_anext</code>	<code>unaryfunc</code>	<code>__anext__</code>
<code>nb_add</code>	<code>binaryfunc</code>	<code>__add__</code> <code>__radd__</code>
<code>nb_inplace_add</code>	<code>binaryfunc</code>	<code>__iadd__</code>
<code>nb_subtract</code>	<code>binaryfunc</code>	<code>__sub__</code> <code>__rsub__</code>
<code>nb_inplace_subtract</code>	<code>binaryfunc</code>	<code>__isub__</code>
<code>nb_multiply</code>	<code>binaryfunc</code>	<code>__mul__</code> <code>__rmul__</code>
<code>nb_inplace_multiply</code>	<code>binaryfunc</code>	<code>__mul__</code>
<code>nb_remainder</code>	<code>binaryfunc</code>	<code>__mod__</code> <code>__rmod__</code>
<code>nb_inplace_remainder</code>	<code>binaryfunc</code>	<code>__mod__</code>
<code>nb_divmod</code>	<code>binaryfunc</code>	<code>__divmod__</code> <code>__rdivmod__</code>
<code>nb_power</code>	<code>ternaryfunc</code>	<code>__pow__</code> <code>__rpow__</code>
<code>nb_inplace_power</code>	<code>ternaryfunc</code>	<code>__pow__</code>
<code>nb_negative</code>	<code>unaryfunc</code>	<code>__neg__</code>
<code>nb_positive</code>	<code>unaryfunc</code>	<code>__pos__</code>
<code>nb_absolute</code>	<code>unaryfunc</code>	<code>__abs__</code>
<code>nb_bool</code>	<code>inquiry</code>	<code>__bool__</code>
<code>nb_invert</code>	<code>unaryfunc</code>	<code>__invert__</code>
<code>nb_lshift</code>	<code>binaryfunc</code>	<code>__lshift__</code> <code>__rlshift__</code>
<code>nb_inplace_lshift</code>	<code>binaryfunc</code>	<code>__lshift__</code>
<code>nb_rshift</code>	<code>binaryfunc</code>	<code>__rshift__</code> <code>__rrshift__</code>
<code>nb_inplace_rshift</code>	<code>binaryfunc</code>	<code>__rshift__</code>
<code>nb_and</code>	<code>binaryfunc</code>	<code>__and__</code> <code>__rand__</code>
<code>nb_inplace_and</code>	<code>binaryfunc</code>	<code>__and__</code>
<code>nb_xor</code>	<code>binaryfunc</code>	<code>__xor__</code> <code>__rxor__</code>
<code>nb_inplace_xor</code>	<code>binaryfunc</code>	<code>__xor__</code>
<code>nb_or</code>	<code>binaryfunc</code>	<code>__or__</code> <code>__ror__</code>
<code>nb_inplace_or</code>	<code>binaryfunc</code>	<code>__or__</code>
<code>nb_int</code>	<code>unaryfunc</code>	<code>__int__</code>
<code>nb_reserved</code>	<code>void *</code>	
<code>nb_float</code>	<code>unaryfunc</code>	<code>__float__</code>
<code>nb_floor_divide</code>	<code>binaryfunc</code>	<code>__floordiv__</code>
<code>nb_inplace_floor_divide</code>	<code>binaryfunc</code>	<code>__floordiv__</code>

다음 페이지에 계속

일부 슬롯은 일반 어트리뷰트 조회 체인을 통해 효과적으로 상속됨에 유의하십시오.

표 2 - 이전 페이지에서 계속

슬롯	형	특수 메서드
<code>nb_true_divide</code>	<code>binaryfunc</code>	<code>__truediv__</code>
<code>nb_inplace_true_divide</code>	<code>binaryfunc</code>	<code>__truediv__</code>
<code>nb_index</code>	<code>unaryfunc</code>	<code>__index__</code>
<code>nb_matrix_multiply</code>	<code>binaryfunc</code>	<code>__matmul__</code> <code>__rmatmul__</code>
<code>nb_inplace_matrix_multiply</code>	<code>binaryfunc</code>	<code>__matmul__</code>
<code>mp_length</code>	<code>lenfunc</code>	<code>__len__</code>
<code>mp_subscript</code>	<code>binaryfunc</code>	<code>__getitem__</code>
<code>mp_ass_subscript</code>	<code>objobjargproc</code>	<code>__setitem__</code> , <code>__delitem__</code>
<code>sq_length</code>	<code>lenfunc</code>	<code>__len__</code>
<code>sq_concat</code>	<code>binaryfunc</code>	<code>__add__</code>
<code>sq_repeat</code>	<code>ssizeargfunc</code>	<code>__mul__</code>
<code>sq_item</code>	<code>ssizeargfunc</code>	<code>__getitem__</code>
<code>sq_ass_item</code>	<code>ssizeobjargproc</code>	<code>__setitem__</code> , <code>__delitem__</code>
<code>sq_contains</code>	<code>objobjproc</code>	<code>__contains__</code>
<code>sq_inplace_concat</code>	<code>binaryfunc</code>	<code>__iadd__</code>
<code>sq_inplace_repeat</code>	<code>ssizeargfunc</code>	<code>__imul__</code>
<code>bf_getbuffer</code>	<code>getbufferproc()</code>	
<code>bf_releasebuffer</code>	<code>releasebufferproc()</code>	

슬롯 typedef

typedef	매개 변수 형	반환형
<i>allocfunc</i>	<i>PyObject</i> * <i>PyTypeObject</i> * <i>Py_ssize_t</i>	<i>PyObject</i> *
<i>destructor</i>	<i>void</i> *	<i>void</i>
<i>freefunc</i>	<i>void</i> *	<i>void</i>
<i>traverseproc</i>	<i>void</i> * <i>visitproc</i> <i>void</i> *	<i>int</i>
<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> *	<i>int</i>
<i>reprfunc</i>	<i>PyObject</i> *	<i>PyObject</i> *
<i>getattrfunc</i>	<i>PyObject</i> * <i>const char</i> *	<i>PyObject</i> *
<i>setattrfunc</i>	<i>PyObject</i> * <i>const char</i> * <i>PyObject</i> *	<i>int</i>
<i>getattrofunc</i>	<i>PyObject</i> * <i>PyObject</i> *	<i>PyObject</i> *
<i>setattrofunc</i>	<i>PyObject</i> * <i>PyObject</i> * <i>PyObject</i> *	<i>int</i>
<i>descrgetfunc</i>	<i>PyObject</i> * <i>PyObject</i> * <i>PyObject</i> *	<i>PyObject</i> *
194 <i>descrsetfunc</i>	<i>PyObject</i> * <i>PyObject</i> *	<i>int</i> Chapter 12. 객체 구현 지원

자세한 내용은 아래 슬롯 형 *typedef*를 참조하십시오.

12.3.2 PyObject 정의

*PyObject*의 구조체 정의는 Include/object.h에서 찾을 수 있습니다. 참조 편의를 위해, 다음에 정의를 반복합니다:

```
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 */
}
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```

getiterfunc tp_iter;
iternextfunc tp_iternext;

/* Attribute descriptor and subclassing stuff */
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 슬롯

형 객체 구조체는 `PyVarObject` 구조체를 확장합니다. `ob_size` 필드는 동적 형(`type_new()`에 의해 만들어집니다, 일반적으로 `class` 문에서 호출됩니다)에 사용됩니다. `PyType_Type`(메타 형)은 `tp_itemsize`를 초기화함에 유의하십시오, 인스턴스(즉, 형 객체)는 반드시 `ob_size` 필드를 가져야 함을 뜻합니다.

`PyObject* PyObject._ob_next`

`PyObject* PyObject._ob_prev`

이 필드는 매크로 `Py_TRACE_REFS`가 정의됐을 때만 존재합니다. `NULL`로의 초기화는 `PyObject_HEAD_INIT` 매크로에 의해 처리됩니다. 정적으로 할당된 객체의 경우, 이 필드는 항상 `NULL`로 유지됩니다. 동적으로 할당된 객체의 경우, 이 두 필드는 객체를 힙에 있는 모든 라이브 객체의 이중 링크 리스트에 연결하는 데 사용됩니다. 이것은 다양한 디버깅 목적으로 사용될 수 있습니다; 현재 유일한 사용은 환경 변수 `PYTHONDUMPREFS`가 설정될 때 실행이 끝날 때 여전히 존재하는 객체를 인쇄하는 것입니다.

계승:

이 필드는 서브 형에 의해 상속되지 않습니다.

`Py_ssize_t PyObject.ob_refcnt`

이것은 `PyObject_HEAD_INIT` 매크로에 의해 1로 초기화된 형 객체의 참조 횟수입니다. 정적으로 할당된 형 객체의 경우 형의 인스턴스(`ob_type`이 형을 다시 가리키는 객체)는 참조로 카운트되지 않습니다. 그러나 동적으로 할당된 형 객체의 경우, 인스턴스는 참조로 카운트됩니다.

계승:

이 필드는 서브 형에 의해 상속되지 않습니다.

*PyObject** **PyObject.ob_type**

이것은 형의 형, 즉 메타 형(metatype)입니다. `PyObject_HEAD_INIT` 매크로에 대한 인자로 초기화되며, 값은 일반적으로 `&PyType_Type`이어야 합니다. 그러나, (적어도) 윈도우에서 사용 가능해야 하는 동적으로 로드 가능한 확장 모듈의 경우, 컴파일러는 유효한 초기화자가 아니라고 불평합니다. 따라서, 규칙은 `NULL`을 `PyObject_HEAD_INIT` 매크로로 전달하고, 다른 작업을 수행하기 전에 모듈의 초기화 함수 시작에서 필드를 명시적으로 초기화하는 것입니다. 이것은 일반적으로 다음과 같이 수행됩니다:

```
Foo_Type.ob_type = &PyType_Type;
```

형의 인스턴스를 만들기 전에 수행해야 합니다. `PyType_Ready()`는 `ob_type`이 `NULL`인지 확인하고, 그렇다면 베이스 클래스의 `ob_type` 필드로 초기화합니다. `PyType_Ready()`는 0이 아니면 이 필드를 변경하지 않습니다.

계승:

이 필드는 서브 형으로 상속됩니다.

12.3.4 PyVarObject 슬롯

Py_ssize_t **PyVarObject.ob_size**

정적으로 할당된 형 객체의 경우, 0으로 초기화해야 합니다. 동적으로 할당된 형 객체의 경우, 이 필드에는 특별한 내부 의미가 있습니다.

계승:

이 필드는 서브 형에 의해 상속되지 않습니다.

12.3.5 PyTypeObject 슬롯

각 슬롯에는 상속을 설명하는 섹션이 있습니다. 필드가 `NULL`로 설정될 때 `PyType_Ready()`가 값을 설정할 수 있으면, “기본값” 섹션도 있습니다. (`PyBaseObject_Type`과 `PyType_Type`에 설정된 많은 필드가 효과적으로 기본값으로 작동함에 유의하십시오.)

*const char** **PyTypeObject.tp_name**

형 이름이 포함된 NUL-종료 문자열을 가리키는 포인터. 모듈 전역으로 액세스 할 수 있는 형의 경우, 문자열은 전체 모듈 이름, 그 뒤에 점, 그 뒤에 형 이름이어야 합니다; 내장형의 경우, 단지 형 이름이어야 합니다. 모듈이 패키지인 서브 모듈이면, 전체 패키지 이름은 전체 모듈 이름의 일부입니다. 예를 들어, 패키지 P의 서브 패키지 Q에 있는 모듈 M에 정의된 T라는 형은 `tp_name` 초기화자가 "P.Q.M.T"이어야 합니다.

동적으로 할당된 형 객체의 경우, 단지 형 이름이어야 하며, 모듈 이름은 형 디렉터리에 키 '___module___'의 값으로 명시적으로 저장됩니다.

정적으로 할당된 형 객체의 경우, `tp_name` 필드에 점이 있어야 합니다. 마지막 점 이전의 모든 것은 ___module___ 어트리뷰트로 액세스 할 수 있으며, 마지막 점 이후의 모든 것은 ___name___ 어트리뷰트로 액세스 할 수 있습니다.

점이 없으면, 전체 `tp_name` 필드는 ___name___ 어트리뷰트로 액세스 할 수 있으며, ___module___ 어트리뷰트는 정의되지 않습니다 (위에서 설명한 대로, 디렉터리에 명시적으로 설정되지 않는 한). 이것은 여러분의 형을 피클 할 수 없다는 것을 뜻합니다. 또한, pydoc으로 만든 모듈 설명서에 나열되지 않습니다.

이 필드는 `NULL`이 아니어야 합니다. `PyTypeObject()`에서 유일하게 필요한 필드입니다 (잠재적인 `tp_itemsize`를 제외하고).

계승:

이 필드는 서브 형에 의해 상속되지 않습니다.

`Py_ssize_t PyObject.tp_basicsize`

`Py_ssize_t PyObject.tp_itemsize`

이 필드를 사용하면 형 인스턴스의 크기를 바이트 단위로 계산할 수 있습니다.

두 가지 종류의 형이 있습니다: 고정 길이 인스턴스의 형은 0 `tp_itemsize` 필드를 갖고, 가변 길이 인스턴스의 형에는 0이 아닌 `tp_itemsize` 필드가 있습니다. 고정 길이 인스턴스의 형의 경우, 모든 인스턴스는 `tp_basicsize`로 지정되는 같은 크기를 갖습니다.

가변 길이 인스턴스의 형의 경우, 인스턴스에는 `ob_size` 필드가 있어야 하며, 인스턴스 크기는 `tp_basicsize`에 N 곱하기 `tp_itemsize`를 더한 값입니다. 여기서 N은 객체의 “길이”입니다. N 값은 일반적으로 인스턴스의 `ob_size` 필드에 저장됩니다. 예외가 있습니다: 예를 들어, 정수는 음수를 나타내기 위해 음의 `ob_size`를 사용하고, N은 `abs(ob_size)`입니다. 또한 인스턴스 배치에 `ob_size` 필드가 있다고 해서 인스턴스 구조체가 가변 길이라는 뜻은 아닙니다(예를 들어, 리스트 형의 구조체는 고정 길이 인스턴스를 갖지만, 해당 인스턴스에는 의미 있는 `ob_size` 필드가 있습니다).

기본 크기에는 매크로 `PyObject_HEAD`나 `PyObject_VAR_HEAD`(인스턴스 구조체를 선언하는 데 사용한 것)에 의해 선언된 인스턴스의 필드가 포함되며, 이것은 다시 존재한다면 `_ob_prev`와 `_ob_next` 필드도 포함됩니다. 이는 `tp_basicsize`의 초기화자를 얻는 유일하게 올바른 방법은 인스턴스 배치를 선언하는 데 사용되는 구조체에 `sizeof` 연산자를 사용하는 것입니다. 기본 크기에는 GC 헤더 크기가 포함되지 않습니다.

정렬(alignment)에 대한 참고 사항: 가변 길이 항목에 특정 정렬이 필요하다면, `tp_basicsize` 값에서 고려되어야 합니다. 예: 형이 double 배열을 구현하는 형을 가정합시다. `tp_itemsize`는 `sizeof(double)`입니다. `tp_basicsize`가 `sizeof(double)`의 배수가 되도록 하는 것은 프로그램러의 책임입니다(이것이 double의 정렬 요구 사항이라고 가정합니다).

가변 길이 인스턴스가 있는 모든 형의 경우, 이 필드는 NULL이 아니어야 합니다.

계승:

이 필드는 서브 형에 의해 별도로 상속됩니다. 베이스형에 0이 아닌 `tp_itemsize`가 있으면, 일반적으로 서브 형에서 `tp_itemsize`를 다른 0이 아닌 값으로 설정하는 것은 안전하지 않습니다(베이스형의 구현에 따라 다르기는 합니다).

destructor `PyObject.tp_dealloc`

인스턴스 파괴자(destructor) 함수에 대한 포인터. (싱글톤 None과 Ellipsis의 경우처럼) 형이 해당 인스턴스가 할당 해제되지 않도록 보장하지 않는 한, 이 함수를 정의해야 합니다. 함수 서명은 다음과 같습니다:

```
void tp_dealloc(PyObject *self);
```

파괴자 함수는 새로운 참조 횟수가 0일 때 `Py_DECREF()`와 `Py_XDECREF()` 매크로에 의해 호출됩니다. 이 시점에, 인스턴스는 여전히 존재하지만, 이에 대한 참조는 없습니다. 파괴자 함수는 인스턴스가 소유한 모든 참조를 해제하고, (버퍼 할당에 사용된 할당 함수에 해당하는 해제 함수를 사용하여) 인스턴스가 소유한 모든 메모리 버퍼를 해제한 다음, 형의 `tp_free` 함수를 호출해야 합니다. 형의 서브 형을 만들 수 없는 경우(`Py_TPFLAGS_BASETYPE` 플래그 비트가 설정되지 않은 경우) `tp_free`를 거치는 대신 객체 할당 해제기(deallocator)를 직접 호출 할 수 있습니다. 객체 할당 해제기는 인스턴스를 할당하는 데 사용된 것이어야 합니다; 인스턴스가 `PyObject_New()`나 `PyObject_VarNew()`를 사용하여 할당되었으면 일반적으로 `PyObject_Del()`이고, 인스턴스가 `PyObject_GC_New()`나 `PyObject_GC_NewVar()`를 사용하여 할당되었으면 `PyObject_GC_Del()`입니다.

마지막으로, 형이 힙 할당(`Py_TPFLAGS_HEAPTYPE`)이면, 할당 해제기는 형 할당 해제기를 호출한 후 해당 형 객체의 참조 횟수를 줄여야 합니다. 매달린(dangling) 포인터를 피하고자, 이렇게 하는 권장 방법은 다음과 같습니다:

```
static void foo_dealloc(foo_object *self) {
    PyObject *tp = Py_TYPE(self);
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
// free references and buffers here
tp->tp_free(self);
Py_DECREF(tp);
}
```

계승:

이 필드는 서브 형으로 상속됩니다.

Py_ssize_t PyObject.tp_vectorcall_offset

간단한 `tp_call`의 더 효율적인 대안인 벡터콜(`vectorcall`) 프로토콜을 사용하여 객체를 호출하는 것을 구현하는 인스턴스별 함수에 대한 선택적 오프셋입니다.

이 필드는 플래그 `_Py_TPFLAGS_HAVE_VECTORCALL`이 설정되었을 때만 사용됩니다. 그럴 때, 이것은 `vectorcallfunc` 포인터의 인스턴스에서의 오프셋을 포함하는 양의 정수여야 합니다. 서명은 `_PyObject_Vectorcall()`과 같습니다:

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

`vectorcallfunc` 포인터는 0일 수 있으며, 이때 인스턴스는 `_Py_TPFLAGS_HAVE_VECTORCALL`이 설정되지 않은 것처럼 작동합니다: 인스턴스를 호출하면 `tp_call`로 폴백 됩니다.

`_Py_TPFLAGS_HAVE_VECTORCALL`을 설정하는 모든 클래스는 `tp_call`도 설정해야 하고, 해당 동작이 `vectorcallfunc` 함수와 일관되도록 만들어야 합니다. `tp_call`을 `PyVectorcall_Call`로 설정하면 됩니다:

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

`tuple`과 `dict`에 각각 주어진 위치와 키워드 인자로 `callable`의 `vectorcallfunc`를 호출합니다.

이 함수는 `tp_call` 슬롯에서 사용하기 위한 것입니다. `tp_call`로 폴백하지 않으며 현재 `_Py_TPFLAGS_HAVE_VECTORCALL` 플래그를 확인하지 않습니다. 객체를 호출하려면, `PyObject_Call` 함수 중 하나를 대신 사용하십시오.

참고: 힙(`heap`) 형에 벡터콜 프로토콜을 구현하는 것은 권장하지 않습니다. 사용자가 파이썬 코드에서 `__call__`을 설정하면, `tp_call`만 갱신되어 벡터콜 함수와 일치하지 않을 수 있습니다.

참고: `tp_vectorcall_offset` 슬롯의 의미론은 잠정적이며 파이썬 3.9에서 완성될 것으로 예상됩니다. 벡터콜을 사용한다면, 파이썬 3.9에서 코드를 갱신할 준비를 하십시오.

버전 3.8에서 변경: 이 슬롯은 파이썬 2.x의 인쇄 포매팅에 사용되었습니다. 파이썬 3.0에서 3.7에서는, 예약되었고 `tp_print`로 명명되었습니다.

계승:

이 필드는 `tp_call`과 함께 서브 형에 의해 상속됩니다: 서브 형의 `tp_call`이 NULL일 때 서브 형은 베이스형에서 `tp_vectorcall_offset`을 상속합니다.

힙 형(파이썬에서 정의된 서브 클래스를 포함합니다)은 `_Py_TPFLAGS_HAVE_VECTORCALL` 플래그를 상속하지 않음에 유의하십시오.

getattrofunc PyObject.tp_getattro

get-attribute-string 함수에 대한 선택적 포인터.

이 필드는 폐지되었습니다. 정의될 때, `tp_getattro` 함수와 같게 작동하지만, 어트리뷰트 이름을 제공하기 위해 파이썬 문자열 객체 대신 C 문자열을 받아들이는 함수를 가리켜야 합니다.

계승:

그룹: `tp_getattr`, `tp_getattro`

이 필드는 `tp_getattro`와 함께 서브 형에 의해 상속됩니다. 서브 형은 서브 형의 `tp_getattr`과 `tp_getattro`가 모두 NULL일 때 베이스형에서 `tp_getattr`과 `tp_getattro`를 모두 상속합니다.

setattrfunc **PyObject.tp_setattr**

어트리뷰트 설정과 삭제를 위한 함수에 대한 선택적 포인터.

이 필드는 폐지되었습니다. 정의될 때, `tp_setattro` 함수와 함께 작동하지만, 어트리뷰트 이름을 제공하기 위해 파이썬 문자열 객체 대신 C 문자열을 받아들이는 함수를 가리켜야 합니다.

계승:

그룹: `tp_setattr`, `tp_setattro`

이 필드는 `tp_setattro`와 함께 서브 형에 의해 상속됩니다. 서브 형은 서브 형의 `tp_setattr`과 `tp_setattro`가 모두 NULL일 때 베이스형에서 `tp_setattr`과 `tp_setattro`를 모두 상속합니다.

*PyAsyncMethods** **PyObject.tp_as_async**

C 수준에서 어웨이터블과 비동기 이터레이터 프로토콜을 구현하는 객체에만 관련된 필드를 포함하는 추가 구조체에 대한 포인터. 자세한 내용은 비동기 객체 구조체를 참조하십시오.

버전 3.5에 추가: 이전에는 `tp_compare`와 `tp_reserved`라고 했습니다.

계승:

`tp_as_async` 필드는 상속되지 않지만, 포함된 필드는 개별적으로 상속됩니다.

reprfunc **PyObject.tp_repr**

내장 함수 `repr()`을 구현하는 함수에 대한 선택적 포인터.

서명은 `PyObject_Repr()`과 같습니다:

```
PyObject *tp_repr(PyObject *self);
```

함수는 문자열이나 유니코드 객체를 반환해야 합니다. 이상적으로, 이 함수는 `eval()`에 전달될 때 적합한 환경이 주어지면 같은 값을 가진 객체를 반환하는 문자열을 반환해야 합니다. 이것이 가능하지 않으면, '<'로 시작하고 '>'로 끝나는 문자열을 반환해야 하는데, 이 문자열에서 객체의 형과 값을 모두 추론할 수 있어야 합니다.

계승:

이 필드는 서브 형으로 상속됩니다.

기본값:

이 필드를 설정하지 않으면, `<%s object at %p>` 형식의 문자열이 반환됩니다. 여기서 `%s`는 형 이름으로, `%p`는 객체의 메모리 주소로 치환됩니다.

*PyNumberMethods** **PyObject.tp_as_number**

숫자 프로토콜을 구현하는 객체에만 관련된 필드를 포함하는 추가 구조체에 대한 포인터. 이 필드는 숫자 객체 구조체에서 설명합니다.

계승:

`tp_as_number` 필드는 상속되지 않지만, 포함된 필드는 개별적으로 상속됩니다.

*PySequenceMethods** **PyObject.tp_as_sequence**

시퀀스 프로토콜을 구현하는 객체에만 관련된 필드를 포함하는 추가 구조체에 대한 포인터. 이 필드는 시퀀스 객체 구조체에서 설명합니다.

계승:

`tp_as_sequence` 필드는 상속되지 않지만, 포함된 필드는 개별적으로 상속됩니다.

***PyMappingMethods** PyObject.tp_as_mapping**

매핑 프로토콜을 구현하는 객체에만 관련된 필드를 포함하는 추가 구조체에 대한 포인터. 이 필드는 매핑 객체 구조체에서 설명합니다.

계승:

`tp_as_mapping` 필드는 상속되지 않지만, 포함된 필드는 개별적으로 상속됩니다.

***hashfunc* PyObject.tp_hash**

내장 함수 `hash()` 를 구현하는 함수에 대한 선택적 포인터.

서명은 `PyObject_Hash()` 와 같습니다:

```
Py_hash_t tp_hash(PyObject *);
```

-1 값은 정상적인 반환 값으로 반환되지 않아야 합니다; 해시값을 계산하는 동안 에러가 발생하면 함수는 예외를 설정하고 -1을 반환해야 합니다.

이 필드가 설정되지 않으면 (그리고 `tp_richcompare`가 설정되지 않으면), 객체의 해시를 취하려는 시도는 `TypeError`를 발생시킵니다. 이것은 `PyObject_HashNotImplemented()`로 설정하는 것과 같습니다.

이 필드는 부모 형에서 해시 메서드의 상속을 차단하기 위해 `PyObject_HashNotImplemented()`로 명시적으로 설정할 수 있습니다. 이것은 파이썬 수준에서의 `__hash__ = None`과 동등한 것으로 해석되어, `isinstance(o, collections.Hashable)`이 `False`를 올바르게 반환하게 합니다. 반대의 경우도 마찬가지입니다 - 파이썬 수준의 클래스에서 `__hash__ = None`을 설정하면 `tp_hash` 슬롯이 `PyObject_HashNotImplemented()`로 설정됩니다.

계승:

그룹: `tp_hash`, `tp_richcompare`

이 필드는 `tp_richcompare`와 함께 서브 형에 의해 상속됩니다: 서브 형의 `tp_richcompare`와 `tp_hash`가 모두 `NULL`일 때, 서브 형은 `tp_richcompare`와 `tp_hash`를 모두 상속합니다.

***ternaryfunc* PyObject.tp_call**

객체 호출을 구현하는 함수에 대한 선택적 포인터. 객체가 콜러블이 아니면 `NULL`이어야 합니다. 서명은 `PyObject_Call()`과 같습니다:

```
PyObject *tp_call(PyObject *self, PyObject *args, PyObject *kwargs);
```

계승:

이 필드는 서브 형으로 상속됩니다.

***reprfunc* PyObject.tp_str**

내장 연산 `str()` 을 구현하는 함수에 대한 선택적 포인터. (`str`는 이제 형이며, `str()`은 그 형의 생성자를 호출함에 유의하십시오. 이 생성자는 `PyObject_Str()`를 호출하여 실제 작업을 수행하고, `PyObject_Str()`은 이 처리기를 호출합니다.)

서명은 `PyObject_Str()`과 같습니다:

```
PyObject *tp_str(PyObject *self);
```

함수는 문자열이나 유니코드 객체를 반환해야 합니다. 다른 것 중에서도, `print()` 함수에 의해 사용될 표현이기 때문에, 객체의 “친숙한” 문자열 표현이어야 합니다.

계승:

이 필드는 서브 형으로 상속됩니다.

기본값:

이 필드를 설정하지 않으면, 문자열 표현을 반환하기 위해 `PyObject_Repr()` 이 호출됩니다.

getattrofunc **PyTypeObject.tp_getattro**

어트리뷰트 읽기(`get-attribute`) 함수에 대한 선택적 포인터.

서명은 `PyObject_GetAttr()` 과 같습니다:

```
PyObject *tp_getattro(PyObject *self, PyObject *attr);
```

일반적으로 이 필드를 `PyObject_GenericGetAttr()` 로 설정하는 것이 편리합니다, 객체 어트리뷰트를 찾는 일반적인 방법을 구현합니다.

계승:

그룹: `tp_getattr, tp_getattro`

이 필드는 `tp_getattr` 과 함께 서브 형에 의해 상속됩니다: 서브 형의 `tp_getattr` 과 `tp_getattro` 가 모두 NULL 일 때 서브 형은 베이스형에서 `tp_getattr` 과 `tp_getattro` 를 모두 상속합니다.

기본값:

`PyBaseObject_Type` 은 `PyObject_GenericGetAttr()` 을 사용합니다.

setattrofunc **PyTypeObject.tp_setattro**

어트리뷰트 설정과 삭제를 위한 함수에 대한 선택적 포인터.

서명은 `PyObject_SetAttr()` 과 같습니다:

```
PyObject *tp_setattro(PyObject *self, PyObject *attr, PyObject *value);
```

또한, `value` 를 NULL 로 설정하여 어트리뷰트를 삭제하는 것을 반드시 지원해야 합니다. 일반적으로 이 필드를 `PyObject_GenericSetAttr()` 로 설정하는 것이 편리합니다, 객체 어트리뷰트를 설정하는 일반적인 방법을 구현합니다.

계승:

그룹: `tp_setattr, tp_setattro`

이 필드는 `tp_setattr` 과 함께 서브 형에 의해 상속됩니다: 서브 형의 `tp_setattr` 과 `tp_setattro` 가 모두 NULL 일 때, 서브 형은 베이스형에서 `tp_setattr` 과 `tp_setattro` 를 모두 상속합니다.

기본값:

`PyBaseObject_Type` 은 `PyObject_GenericSetAttr()` 을 사용합니다.

*PyBufferProcs** **PyTypeObject.tp_as_buffer**

버퍼 인터페이스를 구현하는 객체에만 관련된 필드를 포함하는 추가 구조체에 대한 포인터. 이 필드는 버퍼 객체 구조체에서 설명합니다.

계승:

`tp_as_buffer` 필드는 상속되지 않지만, 포함된 필드는 개별적으로 상속됩니다.

unsigned long PyTypeObject.tp_flags

이 필드는 다양한 플래그의 비트 마스크입니다. 일부 플래그는 특정 상황에 대한 변형 의미론을 나타냅니다; 다른 것들은 역사적으로 항상 존재하지는 않았던 형 객체(또는 `tp_as_number`, `tp_as_sequence`, `tp_as_mapping` 및 `tp_as_buffer` 를 통해 참조되는 확장 구조체)의 특정 필드가 유효함을 나타내는데 사용됩니다; 이러한 플래그 비트가 없으면, 이것이 보호하는 형 필드에 액세스하지 말아야 하며 대신 0 이나 NULL 값을 갖는 것으로 간주해야 합니다.

계승:

이 필드의 상속은 복잡합니다. 대부분 플래그 비트는 개별적으로 상속됩니다, 즉, 베이스형에 플래그 비트가 설정되어 있으면, 서브 형이 이 플래그 비트를 상속합니다. 확장 구조체와 관련된 플래그 비트는 확장 구조체가 상속되면 엄격하게 상속됩니다, 즉, 플래그 비트의 베이스형의 값이 확장 구조체에 대한 포인터와 함께 서브 형으로 복사됩니다. `Py_TPFLAGS_HAVE_GC` 플래그 비트는 `tp_traverse`와 `tp_clear` 필드와 함께 상속됩니다, 즉, 서브 형에서 `Py_TPFLAGS_HAVE_GC` 플래그 비트가 설정되지 않고 서브 형의 `tp_traverse`와 `tp_clear` 필드가 존재하고 NULL 값을 갖는 경우.

기본값:

`PyObject_Type`은 `Py_TPFLAGS_DEFAULT` | `Py_TPFLAGS_BASETYPE`을 사용합니다.

비트 마스크:

다음 비트 마스크가 현재 정의되어 있습니다; 이들은 | 연산자로 함께 OR 하여 `tp_flags` 필드의 값을 형성할 수 있습니다. 매크로 `PyType_HasFeature()`는 형과 플래그 값 `tp`와 `f`를 취하고 `tp->tp_flags & f`가 0이 아닌지 확인합니다.

Py_TPFLAGS_HEAPTYPE

이 비트는 형 객체 자체가 힙에 할당될 때 설정됩니다, 예를 들어, `PyType_FromSpec()`을 사용하여 동적으로 만들어진 형. 이 경우, 인스턴스의 `ob_type` 필드는 형에 대한 참조로 간주하며, 새 인스턴스가 만들어질 때 형 객체가 INCREMENT되고, 인스턴스가 파괴될 때 DECREMENT됩니다(이는 서브 형의 인스턴스에 적용되지 않습니다; 인스턴스의 `ob_type`이 참조하는 형만 INCREMENT나 DECREMENT됩니다).

계승:

???

Py_TPFLAGS_BASETYPE

이 비트는 형을 다른 형의 베이스형으로 사용할 수 있을 때 설정됩니다. 이 비트가 설정되지 않으면 이 형으로 서브 형을 만들 수 없습니다 (Java의 “final” 클래스와 유사합니다).

계승:

???

Py_TPFLAGS_READY

이 비트는 `PyType_Ready()`에 의해 형 객체가 완전히 초기화될 때 설정됩니다.

계승:

???

Py_TPFLAGS_READYING

이 비트는 `PyType_Ready()`가 형 객체를 초기화하는 동안 설정됩니다.

계승:

???

Py_TPFLAGS_HAVE_GC

이 비트는 객체가 가비지 수집을 지원할 때 설정됩니다. 이 비트가 설정되면, 인스턴스는 `PyObject_GC_New()`를 사용하여 만들어져야 하고 `PyObject_GC_Del()`을 사용하여 파괴되어야 합니다. 순환 가비지 수집 지원 섹션에 추가 정보가 있습니다. 이 비트는 또한 GC 관련 필드 `tp_traverse`와 `tp_clear`가 형 객체에 있음을 암시합니다.

계승:

그룹: `Py_TPFLAGS_HAVE_GC`, `tp_traverse`, `tp_clear`

`Py_TPFLAGS_HAVE_GC` 플래그 비트는 `tp_traverse`와 `tp_clear` 필드와 함께 상속됩니다, 즉, 서브 형에서 `Py_TPFLAGS_HAVE_GC` 플래그 비트가 설정되지 않고 서브 형의 `tp_traverse`와 `tp_clear` 필드가 존재하고 NULL 값을 갖는 경우.

Py_TPFLAGS_DEFAULT

이것은 형 객체와 그 확장 구조체에서 특정 필드의 존재와 관련된 모든 비트의 비트 마스크입니다. 현재, 다음과 같은 필드를 포함합니다: `Py_TPFLAGS_HAVE_STACKLESS_EXTENSION`, `Py_TPFLAGS_HAVE_VERSION_TAG`.

계승:

???

Py_TPFLAGS_METHOD_DESCRIPTOR

이 비트는 객체가 연결되지 않은 메서드 (unbound method) 처럼 동작함을 나타냅니다.

이 플래그가 `type(meth)` 에 설정되면:

- `meth.__get__(obj, cls)(*args, **kwargs)`(`obj`가 `None`이 아닐 때)는 `meth(obj, *args, **kwargs)`와 동등해야 합니다.
- `meth.__get__(None, cls)(*args, **kwargs)`는 `meth(*args, **kwargs)`와 동등해야 합니다.

이 플래그는 `obj.meth()`와 같은 일반적인 메서드 호출에 대한 최적화를 가능하게 합니다: `obj.meth`에 대한 임시 “연결된 메서드 (bound method)” 객체를 만들지 않습니다.

버전 3.8에 추가.

계승:

이 플래그는 힙 형에 의해 상속되지 않습니다. 확장형의 경우, `tp_descr_get`이 상속될 때마다 상속됩니다.

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**

이 플래그는 `PyLong_Check()`와 같은 함수에서 형이 내장형의 서브 클래스인지 신속하게 판별하는 데 사용됩니다; 이러한 특정 검사는 `PyObject_IsInstance()`와 같은 일반 검사보다 빠릅니다. 내장에서 상속된 사용자 정의 형은 `tp_flags`를 적절하게 설정해야 합니다, 그렇지 않으면 그러한 형과 상호 작용하는 코드가 사용되는 검사의 유형에 따라 다르게 작동합니다.

Py_TPFLAGS_HAVE_FINALIZE

이 비트는 `tp_finalize` 슬롯이 형 구조체에 있을 때 설정됩니다.

버전 3.4에 추가.

버전 3.8부터 폐지: 인터프리터는 `tp_finalize` 슬롯이 항상 형 구조체에 있다고 가정하기 때문에, 이 플래그는 더는 필요하지 않습니다.

_Py_TPFLAGS_HAVE_VECTORCALL

이 비트는 클래스가 벡터콜 프로토콜을 구현할 때 설정됩니다. 자세한 내용은 `tp_vectorcall_offset`을 참조하십시오.

계승:

이 비트는 `tp_flags`가 재정의되지 않으면 정적(*static*) 서브 형에 설정됩니다: 서브 형의 `tp_call`이 `NULL`이고 서브 형의 `Py_TPFLAGS_HEAPTYPE`이 설정되지 않을 때 서브 형은 베이스형에서 `_Py_TPFLAGS_HAVE_VECTORCALL`을 상속합니다.

힙 형은 `_Py_TPFLAGS_HAVE_VECTORCALL`을 상속하지 않습니다.

참고: 이 플래그는 잠정적이며 파이썬 3.9에서 다른 이름과 변경된 의미론으로 공개되리라고 봅니다. 벡터콜을 사용하면, 파이썬 3.9 용으로 코드를 갱신할 준비를 하십시오.

버전 3.8에 추가.

`const char* PyObject.tp_doc`

이 형 객체에 대한 독스트링을 제공하는 `NULL`-종료 `C` 문자열에 대한 선택적 포인터. 이는 형과 형의 인스턴스에서 `__doc__` 어트리뷰트로 노출됩니다.

계승:

이 필드는 서브 형에 의해 상속되지 않습니다.

traverseproc `PyObject.tp_traverse`

가비지 수집기의 탐색 함수에 대한 선택적 포인터. `Py_TPFLAGS_HAVE_GC` 플래그 비트가 설정된 경우에만 사용됩니다. 서명은 다음과 같습니다:

```
int tp_traverse(PyObject *self, visitproc visit, void *arg);
```

파이썬의 가비지 수집 체계에 대한 자세한 정보는 섹션 [순환 가비지 수집 지원](#)에서 찾을 수 있습니다.

`tp_traverse` 포인터는 가비지 수집기에서 참조 순환을 감지하는 데 사용됩니다. `tp_traverse` 함수의 일반적인 구현은 단순히 인스턴스가 소유하는 파이썬 객체인 각 인스턴스 멤버에 대해 `Py_VISIT()`를 호출합니다. 예를 들어, 다음은 `_thread` 확장 모듈의 함수 `local_traverse()`입니다:

```
static int
local_traverse(localobject *self, visitproc visit, void *arg)
{
    Py_VISIT(self->args);
    Py_VISIT(self->kw);
    Py_VISIT(self->dict);
    return 0;
}
```

`Py_VISIT()`는 참조 순환에 참여할 수 있는 멤버에 대해서만 호출됨에 유의하십시오. `self->key` 멤버도 있지만, `NULL`이나 파이썬 문자열만 가능해서 참조 순환의 일부가 될 수 없습니다.

반면에, 멤버가 사이클의 일부가 될 수 없다는 것을 알고 있더라도, 디버깅 지원을 위해 `gc` 모듈의 `get_referents()` 함수가 그것을 포함하도록 어쨌거나 방문하고 싶을 수 있습니다.

경고: `tp_traverse`를 구현할 때, 인스턴스가 소유하는(강한 참조를 유지하는) 멤버만 방문해야 합니다. 예를 들어, 객체가 `tp_weaklist` 슬롯을 통해 약한 참조를 지원하면, 인스턴스가 자신에 대한 약한 참조를 직접 소유하지 않기 때문에 링크드 리스트를 지원하는 포인터(`tp_weaklist`가 가리키는 것)를 방문해서는 안됩니다(약한 참조 리스트는 약한 참조 장치를 지원하기 위해 거기에 있습니다. 하지만 인스턴스가 아직 살아 있어도 제거할 수 있어서, 인스턴스는 그 안의 요소에 대한 강한 참조를 갖지 않습니다).

`Py_VISIT()`는 `local_traverse()`의 `visit`와 `arg` 매개 변수가 이 이름일 것을 요구합니다; 다른 이름을 붙이지 마십시오.

계승:

그룹: `Py_TPFLAGS_HAVE_GC`, `tp_traverse`, `tp_clear`

이 필드는 `tp_clear`와 `Py_TPFLAGS_HAVE_GC` 플래그 비트와 함께 서브 형에 의해 상속됩니다. 플래그 비트, `tp_traverse` 및 `tp_clear`가 서브 형에서 모두 0이면 모두 베이스형에서 상속됩니다.

inquiry `PyObject.tp_clear`

가비지 수집기의 정리 함수(clear function)에 대한 선택적 포인터. `Py_TPFLAGS_HAVE_GC` 플래그 비트가 설정된 경우에만 사용됩니다. 서명은 다음과 같습니다:

```
int tp_clear(PyObject *);
```

`tp_clear` 멤버 함수는 가비지 수집기에서 감지한 순환 가비지에서 참조 순환을 끊는 데 사용됩니다. 종합하여, 시스템의 모든 `tp_clear` 함수가 결합하여 모든 참조 순환을 끊어야 합니다. 이것은 미묘합니다, 확신이 서지 않으면 `tp_clear` 함수를 제공하십시오. 예를 들어, 튜플 형은 `tp_clear` 함수를 구현하지 않습니다. 튜플만으로는 참조 순환이 구성될 수 없음을 증명할 수 있기 때문입니다. 따라서 다른 형의 `tp_clear` 함수만으로 튜플을 포함하는 순환을 끊기에 충분해야 합니다. 이것은 그리 자명하지 않으며, `tp_clear`를 구현하지 않아도 좋을 만한 이유는 거의 없습니다.

`tp_clear`의 구현은 다음 예제와 같이 파이썬 객체일 수 있는 자신의 멤버에 대한 인스턴스의 참조를 삭제하고 해당 멤버에 대한 포인터를 NULL로 설정해야 합니다:

```
static int
local_clear(localobject *self)
{
    Py_CLEAR(self->key);
    Py_CLEAR(self->args);
    Py_CLEAR(self->kw);
    Py_CLEAR(self->dict);
    return 0;
}
```

참조 제거는 섬세한 작업이라서 `Py_CLEAR()` 매크로를 사용해야 합니다: 포함된 객체에 대한 포인터가 NULL로 설정될 때까지 포함된 객체에 대한 참조를 감소시키지 않아야 합니다. 이는 참조 횟수를 줄이면 포함된 객체가 버려지게 되어 임의의 파이썬 코드 호출을 포함하는 일련의 교정 활동을 촉발할 수 있기 때문입니다 (포함된 객체와 연관된 파이썬라이저나 약한 참조 콜백으로 인해). 그러한 코드가 `self`를 다시 참조 할 수 있다면, 포함된 객체를 더는 사용할 수 없다는 것을 `self`가 알 수 있도록, 포함된 객체에 대한 포인터가 그 시점에 NULL이 되는 것이 중요합니다. `Py_CLEAR()` 매크로는 안전한 순서로 작업을 수행합니다.

`tp_clear` 함수의 목표는 참조 순환을 끊는 것이기 때문에, 참조 순환에 참여할 수 없는 파이썬 문자열이나 파이썬 정수와 같은 포함된 객체를 정리할 필요는 없습니다. 반면에, 포함된 모든 파이썬 객체를 정리하고, 형의 `tp_dealloc` 함수가 `tp_clear`를 호출하도록 작성하는 것이 편리할 수 있습니다.

파이썬의 가비지 수집 체계에 대한 자세한 정보는 섹션 [순환 가비지 수집 지원](#)에서 찾을 수 있습니다.

계승:

그룹: `Py_TPFLAGS_HAVE_GC`, `tp_traverse`, `tp_clear`

이 필드는 `tp_traverse`와 `Py_TPFLAGS_HAVE_GC` 플래그 비트와 함께 서브 형에 의해 상속됩니다: 플래그 비트, `tp_traverse` 및 `tp_clear`가 서브 형에서 모두 0이면 모두 베이스형에서 상속됩니다.

richcmpfunc `PyObject.tp_richcompare`

풍부한 비교 함수(rich comparison function)에 대한 선택적 포인터. 서명은 다음과 같습니다:

```
PyObject *tp_richcompare(PyObject *self, PyObject *other, int op);
```

첫 번째 매개 변수는 `PyObject`에 의해 정의된 형의 인스턴스임이 보장됩니다.

이 함수는 비교 결과(일반적으로 `Py_True`나 `Py_False`)를 반환해야 합니다. 비교가 정의되어 있지 않으면, `Py_NotImplemented`를 반환하고, 다른 에러가 발생하면 `NULL`을 반환하고 예외 조건을 설정해야 합니다.

다음 상수는 `tp_richcompare`와 `PyObject_RichCompare()`의 세 번째 인자로 사용되도록 정의됩니다:

상수	비교
<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>

풍부한 비교 함수를 쉽게 작성할 수 있도록 다음 매크로가 정의됩니다:

Py_RETURN_RICHCOMPARE (VAL_A, VAL_B, op)

비교 결과에 따라, 함수에서 `Py_True`나 `Py_False`를 반환합니다. `VAL_A`와 `VAL_B`는 C 비교 연산자로 순서를 정할 수 있어야 합니다 (예를 들어, C `int`나 `float`일 수 있습니다). 세 번째 인자는 `PyObject_RichCompare()`에서처럼 요청된 연산을 지정합니다.

반환 값의 참조 횟수가 올바르게 증가합니다.

에러가 발생하면, 예외를 설정하고 함수에서 `NULL`을 반환합니다.

버전 3.7에 추가.

계승:

그룹: `tp_hash`, `tp_richcompare`

이 필드는 `tp_hash`와 함께 서브 형에 의해 상속됩니다. 서브 형의 `tp_richcompare`와 `tp_hash`가 모두 `NULL`이면 서브 형은 `tp_richcompare`와 `tp_hash`를 상속합니다.

기본값:

`PyBaseObject_Type`은 상속될 수 있는 `tp_richcompare` 구현을 제공합니다. 그러나, `tp_hash`만 정의하면, 상속된 함수조차 사용되지 않으며 해당 형의 인스턴스는 비교에 참여할 수 없습니다.

Py_ssize_t PyObject.tp_weaklistoffset

이 형의 인스턴스가 약하게 참조할 수 있으면, 이 필드는 0보다 크고 약한 참조 리스트 헤드의 인스턴스 구조체에서의 오프셋을 포함합니다 (있다면 GC 헤더를 무시하고); 이 오프셋은 `PyObject_ClearWeakRefs()`와 `PyWeakref_*()` 함수에서 사용됩니다. 인스턴스 구조체에는 `NULL`로 초기화되는 `PyObject*` 형의 필드가 포함되어야 합니다.

이 필드를 `tp_weaklist`와 혼동하지 마십시오; 그것은 형 객체 자체에 대한 약한 참조의 리스트 헤드입니다.

계승:

이 필드는 서브 형에 의해 상속되지만, 아래 나열된 규칙을 참조하십시오. 서브 형이 이 오프셋을 재정의할 수 있습니다; 이는 서브 형이 베이스형과 다른 약한 참조 리스트 헤드를 사용함을 의미합니다. 리스트 헤드는 항상 `tp_weaklistoffset`을 통해 발견되므로, 문제가 되지 않습니다.

클래스 문으로 정의된 형에 `__slots__` 선언이 없고, 그것의 베이스형 중 약한 참조 가능한 것이 없으면, 약한 참조 리스트 헤드 슬롯을 인스턴스 배치에 추가하고 해당 슬롯 오프셋의 `tp_weaklistoffset`을 설정하여 해당 형을 약하게 참조할 수 있게 만듭니다.

형의 `__slots__` 선언에 `__weakref__`라는 슬롯이 포함되면, 해당 슬롯은 해당 형의 인스턴스에 대한 약한 참조 리스트 헤드가 되고, 슬롯의 오프셋은 형의 `tp_weaklistoffset`에 저장됩니다.

형의 `__slots__` 선언에 `__weakref__` 라는 슬롯이 없으면, 형은 베이스형에서 `tp_weaklistoffset`을 상속합니다.

getterfunc **PyObject.tp_iter**

객체의 이터레이터를 반환하는 함수에 대한 선택적 포인터. 그 존재는 일반적으로 이 형의 인스턴스가 이터러블이라는 신호입니다(시퀀스는 이 함수 없이도 이터러블일 수 있지만).

이 함수는 `PyObject_GetIter()`와 같은 서명을 갖습니다:

```
PyObject *tp_iter(PyObject *self);
```

계승:

이 필드는 서브 형으로 상속됩니다.

iternextfunc **PyObject.tp_iternext**

이터레이터의 다음 항목을 반환하는 함수에 대한 선택적 포인터. 서명은 다음과 같습니다:

```
PyObject *tp_iternext(PyObject *self);
```

이터레이터가 소진되면 NULL을 반환해야 합니다; `StopIteration` 예외가 설정될 수도, 그렇지 않을 수도 있습니다. 다른 예외가 발생하면, 역시 NULL을 반환해야 합니다. 그 존재는 이 형의 인스턴스가 이터레이터라는 신호입니다.

이터레이터 형은 `tp_iter` 함수도 정의해야 하며, 해당 함수는 (새 이터레이터 인스턴스가 아닌) 이터레이터 인스턴스 자체를 반환해야 합니다.

이 함수는 `PyIter_Next()`와 같은 서명을 갖습니다.

계승:

이 필드는 서브 형으로 상속됩니다.

struct PyMethodDef* PyObject.tp_methods

이 형의 일반 메서드를 선언하는 `PyMethodDef` 구조체의 정적 NULL-종료 배열에 대한 선택적 포인터.

배열의 항목마다, 메서드 디스크립터를 포함하는 형의 디서너리(아래 `tp_dict`를 참조하십시오)에 항목이 추가됩니다.

계승:

이 필드는 서브 형에 의해 상속되지 않습니다(메서드는 다른 메커니즘을 통해 상속됩니다).

struct PyMemberDef* PyObject.tp_members

이 형의 인스턴스의 일반 데이터 멤버(필드나 슬롯)를 선언하는 `PyMemberDef` 구조체의 정적 NULL-종료 배열에 대한 선택적 포인터.

배열의 항목마다, 멤버 디스크립터를 포함하는 형의 디서너리(아래 `tp_dict`를 참조하십시오)에 항목이 추가됩니다.

계승:

이 필드는 서브 형에 의해 상속되지 않습니다(멤버는 다른 메커니즘을 통해 상속됩니다).

struct PyGetSetDef* PyObject.tp_getset

이 형의 인스턴스의 계산된 어트리뷰트를 선언하는 `PyGetSetDef` 구조체의 정적 NULL-종료 배열에 대한 선택적 포인터.

배열의 항목마다, `getset` 디스크립터를 포함하는 형의 디서너리(아래 `tp_dict`를 참조하십시오)에 항목이 추가됩니다.

계승:

이 필드는 서브 형에 의해 상속되지 않습니다(계산된 어트리뷰트는 다른 메커니즘을 통해 상속됩니다).

***PyObject** PyTypeObject.tp_base**

형 속성이 상속되는 베이스형에 대한 선택적 포인터. 이 수준에서는, 단일 상속만 지원됩니다; 다중 상속은 메타 형을 호출하여 형 객체를 동적으로 작성해야 합니다.

참고: 슬롯 초기화에는 전역 초기화 규칙이 적용됩니다. C99에서는 초기화자가 “주소 상수(address constants)”여야 합니다. 포인터로 묵시적으로 변환되는 *PyType_GenericNew()*와 같은 함수 지정자는 유효한 C99 주소 상수입니다.

그러나, *PyBaseObject_Type()*과 같은 정적이지 않은 변수에 적용된 단항 ‘&’ 연산자는 주소 상수를 생성할 필요가 없습니다. 컴파일러는 이를 지원할 수 있으며 (gcc는 지원합니다), MSVC는 지원하지 않습니다. 두 컴파일러 모두 이 특정 동작에서 엄격하게 표준을 준수합니다.

결과적으로, *tp_base*는 확장 모듈의 초기화 함수에서 설정되어야 합니다.

계승:

이 필드는 서브 형에 의해 상속되지 않습니다 (명백히).

기본값:

이 필드의 기본값은 *&PyBaseObject_Type*입니다 (파이썬 프로그래머에게는 *object* 형으로 알려져 있습니다).

***PyObject** PyTypeObject.tp_dict**

형의 딕셔너리는 *PyType_Ready()*에 의해 여기에 저장됩니다.

이 필드는 일반적으로 *PyType_Ready*가 호출되기 전에 NULL로 초기화되어야 합니다; 형의 초기 어트리뷰트를 포함하는 딕셔너리로 초기화될 수도 있습니다. 일단 *PyType_Ready()*가 형을 초기화하면, 형에 대한 추가 어트리뷰트가 (*__add__()*와 같은) 오버로드된 연산에 해당하지 않는 경우에만 이 딕셔너리에 추가될 수 있습니다.

계승:

이 필드는 서브 형에 의해 상속되지 않습니다 (여기에 정의된 어트리뷰트는 다른 메커니즘을 통해 상속됩니다).

기본값:

이 필드가 NULL이면, *PyType_Ready()*는 새 딕셔너리를 할당합니다.

경고: *PyDict_SetItem()*을 사용하거나 다른 식으로 딕셔너리 C-API로 *tp_dict*를 수정하는 것은 안전하지 않습니다.

***descrgetfunc* PyTypeObject.tp_descr_get**

“디스크립터 get” 함수에 대한 선택적 포인터.

함수 서명은 다음과 같습니다:

```
PyObject * tp_descr_get(PyObject *self, PyObject *obj, PyObject *type);
```

계승:

이 필드는 서브 형으로 상속됩니다.

***descrsetfunc* PyTypeObject.tp_descr_set**

디스크립터 값을 설정하고 삭제하기 위한 함수에 대한 선택적 포인터.

함수 서명은 다음과 같습니다:

```
int tp_descr_set(PyObject *self, PyObject *obj, PyObject *value);
```

value 인자는 값을 삭제하기 위해 NULL로 설정됩니다.

계승:

이 필드는 서브 형으로 상속됩니다.

Py_ssize_t PyTypeObject.tp_dictoffset

이 형의 인스턴스에 인스턴스 변수를 포함하는 디렉터리가 있으면, 이 필드는 0이 아니며 인스턴스 변수 디렉터리 형의 인스턴스에서의 오프셋을 포함합니다; 이 오프셋은 `PyObject_GenericGetAttr()` 에서 사용됩니다.

이 필드를 `tp_dict`와 혼동하지 마십시오; 그것은 형 객체 자체의 어트리뷰트에 대한 디렉터리입니다.

이 필드의 값이 0보다 크면, 인스턴스 구조체의 시작으로부터의 오프셋을 지정합니다. 값이 0보다 작으면, 인스턴스 구조체의 끝으로부터의 오프셋을 지정합니다. 음수 오프셋은 사용하기에 더 비싸며, 인스턴스 구조체에 가변 길이 부분이 포함될 때에만 사용해야 합니다. 예를 들어 인스턴스 변수 디렉터리를 `str`이나 `tuple`의 서브 형에 추가하는 데 사용됩니다. 디렉터리가 기본 객체 배치에 포함되어 있지 않더라도, `tp_basicsize` 필드는 이 경우 끝에 추가된 디렉터리를 고려해야 함에 유의하십시오. 포인터 크기가 4바이트인 시스템에서, 디렉터리가 구조체의 맨 끝에 있음을 나타내려면 `tp_dictoffset`을 -4로 설정해야 합니다.

인스턴스의 실제 디렉터리 오프셋은 다음과 같이 음의 `tp_dictoffset`으로 계산할 수 있습니다:

```
dictoffset = tp_basicsize + abs(ob_size)*tp_itemsize + tp_dictoffset
if dictoffset is not aligned on sizeof(void*):
    round up to sizeof(void*)
```

여기서 `tp_basicsize`, `tp_itemsize` 및 `tp_dictoffset`은 형 객체에서 취하고, `ob_size`는 인스턴스에서 취합니다. 정수는 `ob_size`의 부호를 사용하여 숫자의 부호를 저장하므로 절댓값이 사용됩니다. (이 계산을 직접 수행할 필요는 없습니다; `_PyObject_GetDictPtr()` 에서 수행합니다.)

계승:

이 필드는 서브 형에 의해 상속됩니다. 하지만 아래 나열된 규칙을 참조하십시오. 서브 형이 이 오프셋을 재정의할 수 있습니다; 이는 서브 형 인스턴스가 베이스형과는 다른 오프셋에 디렉터리를 저장함을 뜻합니다. 디렉터리는 항상 `tp_dictoffset`을 통해 발견되므로, 문제가 되지 않아야 합니다.

클래스 문으로 정의된 형에 `__slots__` 선언이 없고, 인스턴스 변수 디렉터리를 갖는 베이스형이 없을 때, 디렉터리 슬롯이 인스턴스 배치에 추가되고 `tp_dictoffset`은 해당 슬롯의 오프셋으로 설정됩니다.

클래스 문으로 정의된 형에 `__slots__` 선언이 있으면, 형은 베이스형에서 `tp_dictoffset`을 상속합니다.

(`__slots__` 선언에 `__dict__`라는 슬롯을 추가해도 기대하는 효과는 없고, 단지 혼란을 초래합니다. 그러나 `__weakref__`처럼 기능으로 추가해야 할 수도 있습니다.)

기본값:

이 슬롯에는 기본값이 없습니다. 정적 형의 경우, 이 필드가 NULL이면 인스턴스에 대해 `__dict__`가 만들어지지 않습니다.

initproc PyTypeObject.tp_init

인스턴스 초기화 함수에 대한 선택적 포인터.

이 함수는 클래스의 `__init__()` 메서드에 해당합니다. `__init__()`와 마찬가지로, `__init__()`를 호출하지 않고 인스턴스를 작성할 수 있으며, `__init__()` 메서드를 다시 호출하여 인스턴스를 다시 초기화 할 수 있습니다.

함수 서명은 다음과 같습니다:

```
int tp_init(PyObject *self, PyObject *args, PyObject *kwargs);
```

`self` 인자는 초기화될 인스턴스입니다; `args`와 `kwargs` 인자는 `__init__()` 호출의 위치와 키워드 인자를 나타냅니다.

NULL이 아닐 때, `tp_init` 함수는 형을 호출하여 인스턴스를 정상적으로 만들 때, 형의 `tp_new` 함수가 형의 인스턴스를 반환한 후 호출됩니다. `tp_new` 함수가 원래 형의 서브 형이 아닌 다른 형의 인스턴스를 반환하면, 아무런 `tp_init` 함수도 호출되지 않습니다; `tp_new`가 원래 형의 서브 형 인스턴스를 반환하면, 서브 형의 `tp_init`가 호출됩니다.

성공하면 0을 반환하고, 에러 시에는 -1을 반환하고 예외를 설정합니다.

계승:

이 필드는 서브 형으로 상속됩니다.

기본값:

정적 형의 경우 이 필드에는 기본값이 없습니다.

allocfunc **PyObject.tp_alloc**

인스턴스 할당 함수에 대한 선택적 포인터.

함수 서명은 다음과 같습니다:

```
PyObject *tp_alloc(PyObject *self, Py_ssize_t nitems);
```

계승:

이 필드는 정적 서브 형에 의해 상속되지만, 동적 서브 형(클래스 문으로 만들어진 서브 형)에는 상속되지 않습니다.

기본값:

동적 서브 형의 경우, 이 필드는 표준 힙 할당 전략을 강제하기 위해 항상 `PyType_GenericAlloc()` 으로 설정됩니다.

정적 서브 형의 경우, `PyBaseObject_Type`은 `PyType_GenericAlloc()` 을 사용합니다. 이것이 정적으로 정의된 모든 형에 권장되는 값입니다.

newfunc **PyObject.tp_new**

인스턴스 생성 함수에 대한 선택적 포인터.

함수 서명은 다음과 같습니다:

```
PyObject *tp_new(PyObject *subtype, PyObject *args, PyObject *kwargs);
```

`subtype` 인자는 만들어지고 있는 객체의 형입니다; `args`와 `kwargs` 인자는 형 호출의 위치와 키워드 인자를 나타냅니다. 서브 형은 `tp_new` 함수가 호출되는 형과 같은 필요는 없음에 유의하십시오; 이 형의 서브 형일 수 있습니다(하지만 관련이 없는 형은 아닙니다).

`tp_new` 함수는 객체에 공간을 할당하기 위해 `subtype->tp_alloc(subtype, nitems)` 를 호출해야 하고, 그런 다음 꼭 필요한 만큼만 추가 초기화를 수행해야 합니다. 안전하게 무시하거나 반복할 수 있는 초기화는 `tp_init` 처리기에 배치해야 합니다. 간단한 규칙은, 불변 형의 경우 모든 초기화가 `tp_new`에서 수행되어야 하고, 가변형의 경우 대부분 초기화는 `tp_init`로 미뤄져야 합니다.

계승:

이 필드는 `tp_base`가 NULL이나 `&PyBaseObject_Type`인 정적 형에 의해 상속되지 않는 것을 제외하고 서브 형에 의해 상속됩니다.

기본값:

정적 형의 경우 이 필드에는 기본값이 없습니다. 이는 슬롯이 NULL로 정의되었을 때, 새 인스턴스를 만들기 위해 형을 호출할 수 없음을 뜻합니다; 아마도 팩토리 함수와 같은, 인스턴스를 만드는 다른 방법이 있을 것입니다.

freefunc **PyObject.tp_free**

인스턴스 할당 해제 함수에 대한 선택적 포인터. 서명은 다음과 같습니다:

```
void tp_free(void *self);
```

이 서명과 호환되는 초기화자는 `PyObject_Free()` 입니다.

계승:

이 필드는 정적 서브 형에 의해 상속되지만, 동적 서브 형(클래스 문으로 만들어진 서브 형)에는 상속되지 않습니다.

기본값:

동적 서브 형에서, 이 필드는 `PyType_GenericAlloc()` 과 `Py_TPFLAGS_HAVE_GC` 플래그 비트의 값과 일치하기에 적합한 할당 해제기로 설정됩니다.

정적 서브 형의 경우, `PyBaseObject_Type`은 `PyObject_Del`을 사용합니다.

inquiry **PyObject.tp_is_gc**

가비지 수집기에서 호출되는 함수에 대한 선택적 포인터.

가비지 수집기는 특정 객체가 수집 가능한지를 알아야 합니다. 일반적으로, 객체 형의 `tp_flags` 필드를 보고, `Py_TPFLAGS_HAVE_GC` 플래그 비트를 확인하면 충분합니다. 그러나 일부 형에는 정적과 동적으로 할당된 인스턴스가 혼합되어 있으며, 정적으로 할당된 인스턴스는 수집할 수 없습니다. 이러한 형은 이 함수를 정의해야 합니다; 수집 가능한 인스턴스이면 1을, 수집 불가능한 인스턴스이면 0을 반환해야 합니다. 서명은 다음과 같습니다:

```
int tp_is_gc(PyObject *self);
```

(이것의 유일한 예는 형 자체입니다. 메타 형, `PyType_Type`은 이 함수를 정의하여 정적으로 할당된 형과 동적으로 할당된 형을 구별합니다.)

계승:

이 필드는 서브 형으로 상속됩니다.

기본값:

이 슬롯에는 기본값이 없습니다. 이 필드가 NULL이면, `Py_TPFLAGS_HAVE_GC`가 기능적 동등물로 사용됩니다.

*PyObject** **PyObject.tp_bases**

베이스형의 튜플.

이것은 클래스 문으로 만들어진 형에 대해 설정됩니다. 정적으로 정의된 형의 경우 NULL이어야 합니다.

계승:

이 필드는 상속되지 않습니다.

*PyObject** **PyObject.tp_mro**

형 자체에서 시작하여 object로 끝나는 확장된 베이스형 집합을 포함하는 튜플.

계승:

이 필드는 상속되지 않습니다; `PyType_Ready()`에 의해 새로 계산됩니다.

*PyObject** **PyObject.tp_cache**

사용되지 않습니다. 내부 전용.

계승:

이 필드는 상속되지 않습니다.

*PyObject** **PyTypeObject.tp_subclasses**

서브 클래스에 대한 약한 참조 리스트. 내부 전용.

계승:

이 필드는 상속되지 않습니다.

*PyObject** **PyTypeObject.tp_weaklist**

이 형 객체에 대한 약한 참조를 위한 약한 참조 리스트 헤드. 상속되지 않습니다. 내부 전용.

계승:

이 필드는 상속되지 않습니다.

destructor **PyTypeObject.tp_del**

이 필드는 폐지되었습니다. 대신 *tp_finalize*를 사용하십시오.

unsigned int **PyTypeObject.tp_version_tag**

메서드 캐시에 인덱싱하는 데 사용됩니다. 내부 전용.

계승:

이 필드는 상속되지 않습니다.

destructor **PyTypeObject.tp_finalize**

인스턴스 파이널리제이션 함수에 대한 선택적 포인터. 서명은 다음과 같습니다:

```
void tp_finalize(PyObject *self);
```

*tp_finalize*가 설정되면, 인터프리터는 인스턴스를 파이널라이즈 할 때 이를 한 번 호출합니다. 가비지 수집기(인스턴스가 격리된 참조 순환의 일부인 경우)나 객체가 할당 해제되기 직전에 호출됩니다. 어느 쪽이든, 참조 순환을 끊기 전에 호출되어 정상 상태에 있는 객체를 보도록 보장합니다.

*tp_finalize*는 현재 예외 상태를 변경하지 않아야 합니다; 따라서 사소하지 않은 파이널라이저를 작성하는 권장 방법은 다음과 같습니다:

```
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);
}
```

(상속을 통해서도) 이 필드를 고려하려면, *Py_TPFLAGS_HAVE_FINALIZE* 플래그 비트도 설정해야 합니다.

계승:

이 필드는 서브 형으로 상속됩니다.

버전 3.4에 추가.

더 보기:

“안전한 객체 파이널리제이션”(PEP 442)

나머지 필드는 기능 테스트 매크로 `COUNT_ALLOCS`가 정의된 경우에만 정의되며, 내부 전용입니다. 그것들은 완전성을 위해 여기에서 설명합니다. 이러한 필드는 서브 형에 의해 상속되지 않습니다.

`Py_ssize_t PyObject.tp_allocs`
할당 수.

`Py_ssize_t PyObject.tp_frees`
할당 해제 수.

`Py_ssize_t PyObject.tp_maxalloc`
동시에 할당된 최대 객체 수.

`PyObject* PyObject.tp_prev`
0이 아닌 `tp_allocs` 필드를 가진 이전 형 객체를 가리키는 포인터.

`PyObject* PyObject.tp_next`
0이 아닌 `tp_allocs` 필드를 가진 다음 형 객체를 가리키는 포인터.

또한, 가비지 수집된 파이썬에서, `tp_dealloc`은 객체를 만든 스레드뿐만 아니라, 모든 파이썬 스레드에서 호출될 수 있습니다(객체가 참조 횟수 순환의 일부가 되면, 해당 순환은 모든 스레드에서의 가비지 수집으로 수집될 수 있습니다). `tp_dealloc`이 호출되는 스레드는 GIL(전역 인터프리터 록 - Global Interpreter Lock)을 소유하므로, 파이썬 API 호출에는 문제가 되지 않습니다. 그러나, 파괴되는 중인 객체가 다른 C나 C++ 라이브러리의 객체를 파괴하면, `tp_dealloc`을 호출한 스레드에서 그 객체를 파괴해도 라이브러리의 가정을 위반하지 않는지 주의해야 합니다.

12.3.6 힙 형

전통적으로, C 코드에서 정의된 형은 정적(*static*)입니다. 즉 정적 `PyObject` 구조체는 코드에서 직접 정의되고 `PyObjectReady()`를 사용하여 초기화됩니다.

결과적으로 파이썬에서 정의된 형에 비해 형이 제한됩니다:

- 정적 형은 하나의 베이스로 제한됩니다. 즉, 다중 상속을 사용할 수 없습니다.
- 정적 형 객체(그러나 이들의 인스턴스는 아닙니다)는 불변입니다. 파이썬에서 형 객체의 어트리뷰트를 추가하거나 수정할 수 없습니다.
- 정적 형 객체는 서브 인터프리터에서 공유되므로, 서브 인터프리터 관련 상태를 포함하지 않아야 합니다.

또한, `PyObject`는 안정 ABI의 일부가 아니므로, 정적 형을 사용하는 확장 모듈은 특정 파이썬 부 버전(minor version)에 맞게 컴파일해야 합니다.

정적 형에 대한 대안은 힙 할당된 형(*heap-allocated types*), 또는 짧게 힙 형(*heap types*), 인데 이는 파이썬의 `class` 문으로 작성된 클래스와 밀접한 관련이 있습니다.

`PyObject_Spec` 구조체를 채우고 `PyObject_FromSpecWithBases()`를 호출하면 됩니다.

12.4 숫자 객체 구조체

PyNumberMethods

이 구조체는 객체가 숫자 프로토콜을 구현하는 데 사용하는 함수에 대한 포인터를 담습니다. 각 함수는 숫자 프로토콜 섹션에서 설명하는 유사한 이름의 함수가 사용됩니다.

구조체 정의는 다음과 같습니다:


```
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;

    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;
```

참고: 이항과 삼항 함수는 모든 피연산자의 형을 확인하고, 필요한 변환을 구현해야 합니다 (적어도 피연산자 중 하나는 정의된 형의 인스턴스입니다). 주어진 피연산자에 대해 연산이 정의되지 않으면, 이항과 삼항 함수는 `Py_NotImplemented`를 반환해야 하며, 다른 예외가 발생하면 `NULL`을 반환하고 예외를 설정해야 합니다.

참고: `nb_reserved` 필드는 항상 `NULL`이어야 합니다. 이전에는 `nb_long`라고 했으며, 파이썬 3.0.1에서 이름이 바뀌었습니다.

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 매핑 객체 구조체

PyMappingMethods

이 구조체에는 객체가 매핑 프로토콜을 구현하는 데 사용하는 함수에 대한 포인터를 담습니다. 세 개의 멤버가 있습니다:

lenfunc **PyMappingMethods.mp_length**

이 함수는 *PyMapping_Size()*와 *PyObject_Size()*에서 사용되며, 같은 서명을 갖습니다. 객체에 길이가 정의되어 있지 않으면 이 슬롯을 NULL로 설정할 수 있습니다.

binaryfunc **PyMappingMethods.mp_subscript**

이 함수는 *PyObject_GetItem()*과 *PySequence_GetSlice()*에서 사용되며, *PyObject_GetItem()*과 같은 서명을 갖습니다. *PyMapping_Check()* 함수가 1을 반환하려면, 이 슬롯을 채워야합니다, 그렇지 않으면 NULL일 수 있습니다.

objobjargproc **PyMappingMethods.mp_ass_subscript**

이 함수는 *PyObject_SetItem()*, *PyObject_DelItem()*, *PyObject_SetSlice()* 및 *PyObject_DelSlice()*에서 사용됩니다. *PyObject_SetItem()*과 같은 서명을 갖지만, *v*를 NULL로 설정하여 항목을 삭제할 수도 있습니다. 이 슬롯이 NULL이면, 객체는 항목 대입과 삭제를 지원하지 않습니다.

12.6 시퀀스 객체 구조체

PySequenceMethods

이 구조체는 객체가 시퀀스 프로토콜을 구현하는 데 사용하는 함수에 대한 포인터를 담습니다.

lenfunc **PySequenceMethods.sq_length**

이 함수는 *PySequence_Size()*와 *PyObject_Size()*에서 사용되며, 같은 서명을 갖습니다. 또한 *sq_item*과 *sq_ass_item* 슬롯을 통해 음수 인덱스를 처리하는 데 사용됩니다.

binaryfunc **PySequenceMethods.sq_concat**

이 함수는 *PySequence_Concat()*에서 사용되며 같은 서명을 갖습니다. *nb_add* 슬롯을 통해 숫자 덧셈을 시도한 후, + 연산자에서도 사용됩니다.

ssizeargfunc **PySequenceMethods.sq_repeat**

이 함수는 *PySequence_Repeat()*에서 사용되며 같은 서명을 갖습니다. *nb_multiply* 슬롯을 통해 숫자 곱셈을 시도한 후, * 연산자에서도 사용됩니다.

ssizeargfunc **PySequenceMethods.sq_item**

이 함수는 *PySequence_GetItem()*에서 사용되며 같은 서명을 갖습니다. *mp_subscript* 슬롯을 통해 서브스크립션(subscription)을 시도한 후, *PyObject_GetItem()*에서도 사용됩니다. *PySequence_Check()* 함수가 1을 반환하려면, 이 슬롯을 채워야합니다, 그렇지 않으면 NULL일 수 있습니다.

음의 인덱스는 다음과 같이 처리됩니다: *sq_length* 슬롯이 채워지면, 이를 호출하고 시퀀스 길이를 사용하여 *sq_item*에 전달되는 양의 인덱스를 계산합니다. *sq_length*가 NULL이면, 인덱스는 그대로 함수에 전달됩니다.

ssizeobjargproc **PySequenceMethods.sq_ass_item**

이 함수는 *PySequence_SetItem()*에서 사용되며 같은 서명을 갖습니다. *mp_ass_subscript* 슬롯을 통해 항목 대입과 삭제를 시도한 후, *PyObject_SetItem()*과 *PyObject_DelItem()*에서도 사용됩니다. 객체가 항목 대입과 삭제를 지원하지 않으면 이 슬롯은 NULL로 남겨 둘 수 있습니다.

objobjproc **PySequenceMethods.sq_contains**

이 함수는 *PySequence_Contains()*에서 사용될 수 있으며 같은 서명을 갖습니다. 이 슬롯은 NULL로 남겨 둘 수 있습니다, 이때 *PySequence_Contains()*는 일치하는 것을 찾을 때까지 시퀀스를 단순히 탐색합니다.

binaryfunc PySequenceMethods.sq_inplace_concat

이 함수는 `PySequence_InPlaceConcat()`에서 사용되며 같은 서명을 갖습니다. 첫 번째 피연산자를 수정하고 그것을 반환해야 합니다. 이 슬롯은 NULL로 남겨 둘 수 있으며, 이때 `PySequence_InPlaceConcat()`은 `PySequence_Concat()`으로 폴백 됩니다. `nb_inplace_add` 슬롯을 통해 숫자 제자리 덧셈을 시도한 후, 증분 대입 `+=`에서 사용됩니다.

ssizeargfunc PySequenceMethods.sq_inplace_repeat

이 함수는 `PySequence_InPlaceRepeat()`에서 사용되며 같은 서명을 갖습니다. 첫 번째 피연산자를 수정하고 그것을 반환해야 합니다. 이 슬롯은 NULL로 남겨 둘 수 있으며, 이때 `PySequence_InPlaceRepeat()`은 `PySequence_Repeat()`로 폴백 됩니다. `nb_inplace_multiply` 슬롯을 통해 숫자 제자리 곱셈을 시도한 후, 증분 대입 `*=`에서도 사용됩니다.

12.7 버퍼 객체 구조체

PyBufferProcs

이 구조체는 버퍼 프로토콜에 필요한 함수에 대한 포인터를 담습니다. 프로토콜은 제공자(exporter) 객체가 내부 데이터를 소비자 객체에 노출하는 방법을 정의합니다.

getbufferproc PyBufferProcs.bf_getbuffer

이 함수의 서명은 다음과 같습니다:

```
int (PyObject *exporter, Py_buffer *view, int flags);
```

`view`를 채우기 위해 `exporter`에 대한 `flags`에 지정된 요청을 처리합니다. 포인트 (3) 을 제외하고, 이 함수의 구현은 다음 단계를 반드시 수행해야 합니다:

- (1) 요청을 충족할 수 있는지 확인합니다. 그렇지 않으면, `PyExc_BufferError`를 발생시키고 `view->obj`를 NULL로 설정하고 -1을 반환합니다.
- (2) 요청된 필드를 채웁니다.
- (3) 내보내기 횟수에 대한 내부 카운터를 증가시킵니다.
- (4) `view->obj`를 `exporter`로 설정하고 `view->obj`를 증가시킵니다.
- (5) 0을 반환합니다.

`exporter`가 버퍼 공급자의 체인이나 트리의 일부이면, 두 가지 주요 체계를 사용할 수 있습니다:

- 다시 내보내기: 트리의 각 구성원은 제공자 객체의 역할을 하며 `view->obj`를 자신에 대한 새로운 참조로 설정합니다.
- 리디렉션: 버퍼 요청이 트리의 루트 객체로 리디렉션됩니다. 여기서, `view->obj`는 루트 객체에 대한 새로운 참조가 됩니다.

`view`의 개별 필드는 섹션 [버퍼 구조체](#)에 설명되어 있으며, 제공자가 특정 요청에 응답해야 하는 규칙은 섹션 [버퍼 요청 유형](#)에 있습니다.

`Py_buffer` 구조체에서 가리키는 모든 메모리는 제공자에게 속하며 남은 소비자가 없어질 때까지 유효해야 합니다. `format`, `shape`, `strides`, `suboffsets` 및 `internal`은 소비자에게는 읽기 전용입니다.

`PyBuffer_FillInfo()`는 모든 요청 유형을 올바르게 처리하면서 간단한 바이트열 버퍼를 쉽게 노출할 수 있는 방법을 제공합니다.

`PyObject_GetBuffer()`는 이 함수를 감싸는 소비자 용 인터페이스입니다.

releasebufferproc PyBufferProcs.bf_releasebuffer

이 함수의 서명은 다음과 같습니다:

```
void (PyObject *exporter, Py_buffer *view);
```

버퍼 자원 해제 요청을 처리합니다. 자원을 해제할 필요가 없으면, `PyBufferProcs.bf_releasebuffer`는 NULL일 수 있습니다. 그렇지 않으면, 이 함수의 표준 구현은 다음과 같은 선택적 단계를 수행합니다:

- (1) 내보내기 횟수에 대한 내부 카운터를 줄입니다.
- (2) 카운터가 0이면, `view`와 관련된 모든 메모리를 해제합니다.

제공자는 반드시 `internal` 필드를 사용하여 버퍼 특정 자원을 추적해야 합니다. 이 필드는 변경되지 않고 유지됨이 보장되지만, 소비자는 원래 버퍼의 사본을 `view` 인자로 전달할 수 있습니다.

이 함수는 `PyBuffer_Release()`에서 자동으로 수행되므로 `view->obj`를 절대 감소시키지 않아야 합니다(이 체계는 참조 순환을 끊는 데 유용합니다).

`PyBuffer_Release()`는 이 기능을 감싸는 소비자 용 인터페이스입니다.

12.8 비동기 객체 구조체

버전 3.5에 추가.

PyAsyncMethods

이 구조체는 어웨이터블과 비동기 이터레이터 객체를 구현하는 데 필요한 함수에 대한 포인터를 담습니다.

구조체 정의는 다음과 같습니다:

```
typedef struct {
    unaryfunc am_await;
    unaryfunc am_aiter;
    unaryfunc am_anext;
} PyAsyncMethods;
```

`unaryfunc PyAsyncMethods.am_await`

이 함수의 서명은 다음과 같습니다:

```
PyObject *am_await(PyObject *self);
```

반환된 객체는 이터레이터여야 합니다, 즉, `PyIter_Check()`는 반환된 객체에 대해 1을 반환해야 합니다.

객체가 어웨이터블이 아니면 이 슬롯을 NULL로 설정할 수 있습니다.

`unaryfunc PyAsyncMethods.am_aiter`

이 함수의 서명은 다음과 같습니다:

```
PyObject *am_aiter(PyObject *self);
```

어웨이터블 객체를 반환해야 합니다. 자세한 내용은 `__anext__()`를 참조하십시오.

객체가 비동기 이터레이션 프로토콜을 구현하지 않으면 이 슬롯은 NULL로 설정될 수 있습니다.

`unaryfunc PyAsyncMethods.am_anext`

이 함수의 서명은 다음과 같습니다:

```
PyObject *am_anext(PyObject *self);
```

어웨이터블 객체를 반환해야 합니다. 자세한 내용은 `__anext__()` 를 참조하십시오. 이 슬롯은 NULL로 설정될 수 있습니다.

12.9 슬롯 형 typedef

PyObject * (***allocfunc**) (*PyObject* *cls, Py_ssize_t nitems)

이 함수의 목적은 메모리 초기화에서 메모리 할당을 분리하는 것입니다. 인스턴스에 적합한 길이의, 적절하게 정렬되고, 0으로 초기화되지만, `ob_refcnt`는 1로 설정되고 `ob_type`은 형 인자로 설정된 메모리 블록에 대한 포인터를 반환해야 합니다. 형의 `tp_itemsize`가 0이 아니면, 객체의 `ob_size` 필드는 `nitems`로 초기화되고 할당된 메모리 블록의 길이는 `tp_basicsize + nitems*tp_itemsize` 여야 하는데, `sizeof(void*)`의 배수로 자리 올림 되어야 합니다; 그렇지 않으면 `nitems`가 사용되지 않으며 블록의 길이는 `tp_basicsize` 여야 합니다.

이 함수는 다른 인스턴스 초기화를 수행하지 않아야 합니다, 추가 메모리를 할당도 안 됩니다; 그것은 `tp_new`에 의해 수행되어야 합니다.

void (***destructor**) (*PyObject* *)

PyObject * (***vectorcallfunc**) (*PyObject* *callable, *PyObject* *const *args, size_t nargsf, *PyObject* *kw-names)

`tp_vectorcall_offset`을 참조하십시오.

`vectorcallfunc`에 대한 인자는 `_PyObject_Vectorcall()`과 같습니다.

버전 3.8에 추가.

void (***freefunc**) (void *)

`tp_free`를 참조하십시오.

PyObject * (***newfunc**) (*PyObject* *, *PyObject* *, *PyObject* *)

`tp_new`를 참조하십시오.

int (***initproc**) (*PyObject* *, *PyObject* *, *PyObject* *)

`tp_init`를 참조하십시오.

PyObject * (***reprfunc**) (*PyObject* *)

`tp_repr`를 참조하십시오.

PyObject * (***getattrfunc**) (*PyObject* *self, char *attr)

객체의 명명된 어트리뷰트 값을 반환합니다.

int (***setattrfunc**) (*PyObject* *self, char *attr, *PyObject* *value)

객체의 명명된 어트리뷰트 값을 설정합니다. 어트리뷰트를 삭제하려면 `value` 인자가 NULL로 설정됩니다.

PyObject * (***getattrofunc**) (*PyObject* *self, *PyObject* *attr)

객체의 명명된 어트리뷰트 값을 반환합니다.

`tp_getattro`를 참조하십시오.

int (***setattrofunc**) (*PyObject* *self, *PyObject* *attr, *PyObject* *value)

객체의 명명된 어트리뷰트 값을 설정합니다. 어트리뷰트를 삭제하려면 `value` 인자가 NULL로 설정됩니다.

`tp_setattro`를 참조하십시오.

PyObject * (***descrgetfunc**) (*PyObject* *, *PyObject* *, *PyObject* *)

`tp_descrget`를 참조하십시오.

int (***descrsetfunc**) (*PyObject* *, *PyObject* *, *PyObject* *)

`tp_descrset`를 참조하십시오.


```

Py_hash_t (*hashfunc) (PyObject *)
    tp_hash를 참조하십시오.

PyObject* (*richcmpfunc) (PyObject *, PyObject *, int)
    tp_richcompare를 참조하십시오.

PyObject* (*getiterfunc) (PyObject *)
    tp_iter를 참조하십시오.

PyObject* (*iternextfunc) (PyObject *)
    tp_iternext를 참조하십시오.

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 예

다음은 파이썬 형 정의의 간단한 예입니다. 여기에는 여러분이 만날 수 있는 일반적인 사용법이 포함됩니다. 일부는 까다로운 코너 사례를 보여줍니다. 더 많은 예제, 실용 정보 및 자습서는 `defining-new-types`와 `new-types-topics`를 참조하십시오.

기본 정적 형:

```

typedef struct {
    PyObject_HEAD
    const char *data;
} PyObject;

static PyTypeObject PyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyObject",
    .tp_basicsize = sizeof(PyObject),
    .tp_doc = "My objects",
    .tp_new = myobj_new,
    .tp_dealloc = (destructor)myobj_dealloc,
    .tp_repr = (reprfunc)myobj_repr,
};

```

더 상세한 초기화자를 사용하는 이전 코드(특히 CPython 코드 베이스에서)를 찾을 수도 있습니다:

```

static PyTypeObject PyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    "mymod.MyObject",          /* tp_name */

```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```

sizeof(MyObject),          /* tp_basicsize */
0,                          /* tp_itemsize */
(destructor)myobj_dealloc, /* tp_dealloc */
0,                          /* tp_vectorcall_offset */
0,                          /* tp_getattr */
0,                          /* tp_setattr */
0,                          /* tp_as_async */
(reprfunc)myobj_repr,      /* tp_repr */
0,                          /* tp_as_number */
0,                          /* tp_as_sequence */
0,                          /* tp_as_mapping */
0,                          /* tp_hash */
0,                          /* tp_call */
0,                          /* tp_str */
0,                          /* tp_getattro */
0,                          /* tp_setattro */
0,                          /* tp_as_buffer */
0,                          /* tp_flags */
"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 */
};

```

약한 참조, 인스턴스 디렉터리 및 해싱을 지원하는 형:

```

typedef struct {
    PyObject_HEAD
    const char *data;
    PyObject *inst_dict;
    PyObject *weakreflist;
} MyObject;

static PyTypeObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyObject",
    .tp_basicsize = sizeof(MyObject),
    .tp_doc = "My objects",
    .tp_weaklistoffset = offsetof(MyObject, weakreflist),
    .tp_dictoffset = offsetof(MyObject, inst_dict),
    .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE | Py_TPFLAGS_HAVE_GC,
    .tp_new = myobj_new,

```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```

    .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,
};

```

서브 클래싱 할 수 없고 인스턴스를 만들기 위해 호출할 수 없는 str 서브 클래스 (예를 들어 별도의 팩토리 함수를 사용합니다):

```

typedef struct {
    PyUnicodeObject raw;
    char *extra;
} MyStr;

static PyTypeObject MyStr_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyStr",
    .tp_basicsize = sizeof(MyStr),
    .tp_base = NULL, // set to &PyUnicode_Type in module init
    .tp_doc = "my custom str",
    .tp_flags = Py_TPFLAGS_DEFAULT,
    .tp_new = NULL,
    .tp_repr = (reprfunc)myobj_repr,
};

```

(고정 길이 인스턴스의) 가장 간단한 정적 형:

```

typedef struct {
    PyObject_HEAD
} MyObject;

static PyTypeObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyObject",
};

```

(가변 길이 인스턴스의) 가장 간단한 정적 형:

```

typedef struct {
    PyObject_VAR_HEAD
    const char *data[1];
} MyObject;

static PyTypeObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyObject",
    .tp_basicsize = sizeof(MyObject) - sizeof(char *),
    .tp_itemsize = sizeof(char *),
};

```

12.11 순환 가비지 수집 지원

순환 참조를 포함하는 가비지를 탐지하고 수집하는 파이썬의 지원은 역시 컨테이너일 수 있는 다른 객체의 “컨테이너”인 객체 형의 지원이 필요합니다. 다른 객체에 대한 참조를 저장하지 않거나, 원자형(가령 숫자나 문자열)에 대한 참조만 저장하는 형은 가비지 수집에 대한 어떤 명시적인 지원을 제공할 필요가 없습니다.

컨테이너형을 만들려면, 형 객체의 `tp_flags` 필드가 `Py_TPFLAGS_HAVE_GC`를 포함해야 하고 `tp_traverse` 처리기 구현을 제공해야 합니다. 형의 인스턴스가 가변이면, `tp_clear` 구현도 제공해야 합니다.

Py_TPFLAGS_HAVE_GC

이 플래그가 설정된 형의 객체는 여기에 설명된 규칙을 준수해야 합니다. 편의를 위해 이러한 객체를 컨테이너 객체라고 하겠습니다.

컨테이너형의 생성자는 두 가지 규칙을 준수해야 합니다:

1. 객체의 메모리는 `PyObject_GC_New()` 나 `PyObject_GC_NewVar()`를 사용하여 할당해야 합니다.
2. 다른 컨테이너에 대한 참조를 포함할 수 있는 모든 필드가 초기화되면, `PyObject_GC_Track()`를 호출해야 합니다.

TYPE* **PyObject_GC_New**(TYPE, PyTypeObject *type)

`PyObject_New()`와 유사하지만, `Py_TPFLAGS_HAVE_GC` 플래그가 설정된 컨테이너 객체를 위한 것.

TYPE* **PyObject_GC_NewVar**(TYPE, PyTypeObject *type, Py_ssize_t size)

`PyObject_NewVar()`와 유사하지만, `Py_TPFLAGS_HAVE_GC` 플래그가 설정된 컨테이너 객체를 위한 것.

TYPE* **PyObject_GC_Resize**(TYPE, PyVarObject *op, Py_ssize_t newsize)

`PyObject_NewVar()`에 의해 할당된 객체의 크기를 변경합니다. 크기가 조정된 객체나 실패하면 NULL을 반환합니다. `op`는 아직 수집기가 추적하지 않아야 합니다.

void **PyObject_GC_Track**(PyObject *op)

수집기가 추적하는 컨테이너 객체 집합에 객체 `op`를 추가합니다. 수집기는 예기치 않은 시간에 실행될 수 있으므로 추적되는 동안 객체가 유효해야 합니다. `tp_traverse` 처리기가 탐색하는 모든 필드가 유효해지면 호출해야 합니다, 보통 생성자의 끝부분 근처입니다.

마찬가지로, 객체의 할당해제자(deallocator)는 비슷한 규칙 쌍을 준수해야 합니다:

1. 다른 컨테이너를 참조하는 필드가 무효화 되기 전에, `PyObject_GC_UnTrack()`를 호출해야 합니다.
2. 객체의 메모리는 `PyObject_GC_Del()`를 사용하여 할당 해제되어야 합니다.

void **PyObject_GC_Del**(void *op)

`PyObject_GC_New()`나 `PyObject_GC_NewVar()`를 사용하여 객체에 할당된 메모리를 해제합니다.

void **PyObject_GC_UnTrack**(void *op)

수집기가 추적하는 컨테이너 객체 집합에서 `op` 객체를 제거합니다. `PyObject_GC_Track()`를 이 객체에 대해 다시 호출하여 추적 객체 집합에 다시 추가할 수 있음에 유의하십시오. 할당해제자(`tp_dealloc` 처리기)는 `tp_traverse` 처리기에서 사용하는 필드가 무효화 되기 전에 객체에 대해 이 함수를 호출해야 합니다.

버전 3.8에서 변경: `_PyObject_GC_TRACK()`과 `_PyObject_GC_UNTRACK()` 매크로는 공용 C API에서 제거되었습니다.

`tp_traverse` 처리기는 다음과 같은 형의 함수 매개 변수를 받아들입니다:

int (***visitproc**)(PyObject *object, void *arg)

`tp_traverse` 처리기에 전달되는 방문자 함수의 형. 이 함수는 탐색하는 객체를 `object`로, `tp_traverse` 처리기의 세 번째 매개 변수를 `arg`로 호출되어야 합니다. 파이썬 코어는 순환 가비지 탐지를 구현하기 위해 여러 방문자 함수를 사용합니다; 사용자가 자신의 방문자 함수를 작성해야 할 필요는 없습니다.

`tp_traverse` 처리기는 다음 형이어야 합니다:

`int (*traverseproc) (PyObject *self, visitproc visit, void *arg)`

컨테이너 객체의 탐색 함수입니다. 구현은 `self`에 직접 포함된 각 객체에 대해 `visit` 함수를 호출해야 하며, `visit`에 대한 매개 변수는 포함된 객체와 처리기로 전달된 `arg` 값입니다. `visit` 함수는 NULL object 인자로 호출하면 안 됩니다. `visit`가 0이 아닌 값을 반환하면 그 값이 즉시 반환되어야 합니다.

`tp_traverse` 처리기 작성을 단순화하기 위해, `Py_VISIT()` 매크로가 제공됩니다. 이 매크로를 사용하려면, `tp_traverse` 구현은 인자의 이름을 정확히 `visit` 와 `arg`로 지정해야 합니다:

`void Py_VISIT (PyObject *o)`

`o`가 NULL이 아니면, `o` 와 `arg` 인자로 `visit` 콜백을 호출합니다. `visit`가 0이 아닌 값을 반환하면, 그것을 반환합니다. 이 매크로를 사용하면, `tp_traverse` 처리기가 다음과 같아집니다:

```
static int
my_traverse(Noddy *self, visitproc visit, void *arg)
{
    Py_VISIT(self->foo);
    Py_VISIT(self->bar);
    return 0;
}
```

`tp_clear` 처리기는 `inquiry` 형이거나 객체가 불변이면 NULL이어야 합니다.

`int (*inquiry) (PyObject *self)`

참조 순환을 생성했을 수 있는 참조를 삭제합니다. 불변 객체는 참조 순환을 직접 생성할 수 없으므로, 이 메서드를 정의 할 필요가 없습니다. 이 메서드를 호출한 후에도 객체가 유효해야 합니다 (단지 참조에 대해 `Py_DECREF()` 를 호출하지 마십시오). 이 객체가 참조 순환에 참여하고 있음을 수집기가 감지하면 이 메서드를 호출합니다.

CHAPTER 13

API와 ABI 버전 붙이기

PY_VERSION_HEX는 단일 정수로 인코딩된 파이썬 버전 번호입니다.

예를 들어 PY_VERSION_HEX가 0x030401a2로 설정되면, 기본 버전 정보는 다음과 같은 방식으로 32비트 숫자로 처리하여 찾을 수 있습니다:

바이트	비트 (빅 엔디안 순서)	뜻
1	1-8	PY_MAJOR_VERSION (3.4.1a2의 3)
2	9-16	PY_MINOR_VERSION (3.4.1a2의 4)
3	17-24	PY_MICRO_VERSION (3.4.1a2의 1)
4	25-28	PY_RELEASE_LEVEL (알파는 0xA, 베타는 0xB, 배포 후보는 0xC, 최종은 0xF). 이 예에서는 알파입니다.
	29-32	PY_RELEASE_SERIAL (3.4.1a2의 2, 최종 배포는 0)

따라서 3.4.1a2는 16진수 버전 0x030401a2입니다.

모든 주어진 매크로는 `Include/patchlevel.h`에 정의됩니다.

>>> 대화형 셸의 기본 파이썬 프롬프트. 인터프리터에서 대화형으로 실행될 수 있는 코드 예에서 자주 볼 수 있습니다.

... 다음과 같은 것들을 가리킬 수 있습니다:

- 들여쓰기 된 코드 블록의 코드를 입력할 때, 쌍을 이루는 구분자(괄호, 대괄호, 중괄호) 안에 코드를 입력할 때, 데코레이터 지정 후의 대화형 셸의 기본 파이썬 프롬프트.
- Ellipsis 내장 상수.

2to3 파이썬 2.x 코드를 파이썬 3.x 코드로 변환하려고 시도하는 도구인데, 소스를 구문 분석하고 구문 분석 트리를 탐색해서 감지할 수 있는 대부분의 비호환성을 다룹니다.

2to3 는 표준 라이브러리에서 lib2to3 로 제공됩니다; 독립적으로 실행할 수 있는 스크립트는 Tools/scripts/2to3 로 제공됩니다. 2to3-reference 을 보세요.

abstract base class (추상 베이스 클래스) 추상 베이스 클래스는 `hasattr()` 같은 다른 테크닉들이 불편하거나 미묘하게 잘못된 (예를 들어, 매직 메서드) 경우, 인터페이스를 정의하는 방법을 제공함으로써 **덕 타이핑** 을 보완합니다. ABC는 가상 서브 클래스를 도입하는데, 클래스를 계승하지 않으면서도 `isinstance()` 와 `issubclass()` 에 의해 감지될 수 있는 클래스들입니다; abc 모듈 설명서를 보세요. 파이썬에는 많은 내장 ABC 들이 따라오는데 다음과 같은 것들이 있습니다: 자료 구조(`collections.abc` 모듈에서), 숫자(`numbers` 모듈에서), 스트림(`io` 모듈에서), 임포트 과인더와 로더(`importlib.abc` 모듈에서). abc 모듈을 사용해서 자신만의 ABC를 만들 수도 있습니다.

annotation (어노테이션) 관습에 따라 **형 힌트** 로 사용되는 변수, 클래스 어트리뷰트 또는 함수 매개변수 나 반환 값과 연결된 레이블입니다.

지역 변수의 어노테이션은 실행 시간에 액세스할 수 없지만, 전역 변수, 클래스 속성 및 함수의 어노테이션은 각각 모듈, 클래스, 함수의 `__annotations__` 특수 어트리뷰트에 저장됩니다.

이 기능을 설명하는 **변수 어노테이션**, **함수 어노테이션**, **PEP 484**, **PEP 526**을 참조하세요.

argument (인자) 함수를 호출할 때 함수 (또는 메서드) 로 전달되는 값. 두 종류의 인자가 있습니다:

- 키워드 인자 (*keyword argument*): 함수 호출 때 식별자가 앞에 붙은 인자 (예를 들어, `name=`) 또는 `**` 를 앞에 붙인 딕셔너리로 전달되는 인자. 예를 들어, 다음과 같은 `complex()` 호출에서 3 과 5 는 모두 키워드 인자입니다:

```
complex(real=3, imag=5)
complex(**{'real': 3, 'imag': 5})
```

- 위치 인자 (*positional argument*): 키워드 인자가 아닌 인자. 위치 인자들은 인자 목록의 처음에 나오거나 *이터러블*의 앞에 *를 붙여 전달할 수 있습니다. 예를 들어, 다음과 같은 호출에서 3과 5는 모두 위치 인자입니다.

```
complex(3, 5)
complex(*(3, 5))
```

인자는 함수 바디의 이름 붙은 지역 변수에 대입됩니다. 이 대입에 적용되는 규칙들에 대해서는 [calls](#) 절을 보세요. 문법적으로, 어떤 표현식이건 인자로 사용될 수 있습니다; 구해진 값이 지역 변수에 대입됩니다.

용어집의 [매개변수](#) 항목과 FAQ 질문 인자와 매개변수의 차이와 [PEP 362](#)도 보세요.

asynchronous context manager (비동기 컨텍스트 관리자) `__aenter__()`와 `__aexit__()` 메서드를 정의함으로써 `async with` 문에서 보이는 환경을 제어하는 객체. [PEP 492](#)로 도입되었습니다.

asynchronous generator (비동기 제너레이터) 비동기 제너레이터 *이터레이터*를 돌려주는 함수. `async def`로 정의되는 코루틴 함수처럼 보이는데, `async for` 루프가 사용할 수 있는 일련의 값들을 만드는 `yield` 표현식을 포함한다는 점이 다릅니다.

보통 비동기 제너레이터 함수를 가리키지만, 어떤 문맥에서는 비동기 제너레이터 *이터레이터*를 가리킵니다. 의도하는 의미가 명확하지 않은 경우는, 완전한 용어를 써서 모호함을 없앱니다.

비동기 제너레이터 함수는 `await` 표현식과, `async for` 문과, `async with` 문을 포함할 수 있습니다.

asynchronous generator iterator (비동기 제너레이터 *이터레이터*) 비동기 제너레이터 함수가 만드는 객체.

비동기 *이터레이터* 인데 `__anext__()`를 호출하면 어웨이터블 객체를 돌려주고, 이것은 다음 `yield` 표현식까지 비동기 제너레이터 함수의 바디를 실행합니다.

각 `yield`는 일시적으로 처리를 중단하고, 그 위치의 (지역 변수들과 대기 중인 `try`-문들을 포함하는) 실행 상태를 기억합니다. 비동기 제너레이터 *이터레이터*가 `__anext__()`가 돌려주는 또 하나의 어웨이터블로 재개되면, 떠난 곳으로 복귀합니다. [PEP 492](#)와 [PEP 525](#)를 보세요.

asynchronous iterable (비동기 *이터러블*) `async for` 문에서 사용될 수 있는 객체. `__aiter__()` 메서드는 비동기 *이터레이터*를 돌려줘야 합니다. [PEP 492](#)로 도입되었습니다.

asynchronous iterator (비동기 *이터레이터*) `__aiter__()`와 `__anext__()` 메서드를 구현하는 객체. `__anext__`는 어웨이터블 객체를 돌려줘야 합니다. `async for`는 `StopAsyncIteration` 예외가 발생할 때까지 비동기 *이터레이터*의 `__anext__()` 메서드가 돌려주는 어웨이터블을 팝니다. [PEP 492](#)로 도입되었습니다.

attribute (어트리뷰트) 점표현식을 사용하는 이름으로 참조되는 객체와 결합한 값. 예를 들어, 객체 *o*가 어트리뷰트 *a*를 가지면, *o.a*처럼 참조됩니다.

awaitable (어웨이터블) `await` 표현식에 사용할 수 있는 객체. 코루틴이나 `__await__()` 메서드를 가진 객체가 될 수 있습니다. [PEP 492](#)를 보세요.

BDFL 자비로운 종신 독재자 (Benevolent Dictator For Life), 즉 [Guido van Rossum](#), 파이썬의 창시자.

binary file (바이너리 파일) 바이트열류 객체들을 읽고 쓸 수 있는 파일 객체. 바이너리 파일의 예로는 바이너리 모드 ('rb', 'wb' 또는 'rb+')로 열린 파일, `sys.stdin.buffer`, `sys.stdout.buffer`, `io.BytesIO`와 `gzip.GzipFile`의 인스턴스를 들 수 있습니다.

`str` 객체를 읽고 쓸 수 있는 파일 객체에 대해서는 [텍스트 파일](#)도 참조하세요.

bytes-like object (바이트열류 객체) *버퍼 프로토콜*을 지원하고 C-연속 버퍼를 익스포트 할 수 있습니다. 여러 공통 `memoryview` 객체들은 물론이고 `bytes`, `bytearray`, `array.array` 객체들을 포함합니다. 바이트열류 객체들은 바이너리 데이터를 다루는 여러 가지 연산들에 사용될 수 있습니다; 압축, 바이너리 파일로 저장, 소켓을 통한 전송 같은 것들이 있습니다.

어떤 연산들은 바이너리 데이터가 가변적일 필요가 있습니다. 이런 경우에 설명서는 종종 “읽고-쓰기 바이트열류 객체”라고 표현합니다. 가변 버퍼 객체의 예로는 `bytearray`와 `bytearray`의 `memoryview`가 있습니다. 다른 연산들은 바이너리 데이터가 불변 객체 (“읽기 전용 바이트열류 객체”)에 저장되도록 요구합니다; 이런 것들의 예로는 `bytes`와 `bytes` 객체의 `memoryview`가 있습니다.

bytecode (바이트 코드) 파이썬 소스 코드는 바이트 코드로 컴파일되는데, CPython 인터프리터에서 파이썬 프로그램의 내부 표현입니다. 바이트 코드는 `.pyc` 파일에 캐시 되어, 같은 파일을 두 번째 실행할 때 더 빨라지게 만듭니다 (소스에서 바이트 코드로의 재컴파일을 피할 수 있습니다). 이 “중간 언어”는 각 바이트 코드에 대응하는 기계를 실행하는 **가상 기계**에서 실행된다고 말합니다. 바이트 코드는 서로 다른 파이썬 가상 기계에서 작동할 것으로 기대하지도, 파이썬 배포 간에 안정적이지도 않다는 것에 주의해야 합니다.

바이트 코드 명령어들의 목록은 `dis` 모듈 설명서에 나옵니다.

callback (콜백) 인자로 전달되는 미래의 어느 시점에서 실행될 서브 루틴 함수.

class (클래스) 사용자 정의 객체들을 만들기 위한 주형. 클래스 정의는 보통 클래스의 인스턴스를 대상으로 연산하는 메서드 정의들을 포함합니다.

class variable (클래스 변수) 클래스에서 정의되고 클래스 수준 (즉, 클래스의 인스턴스에서가 아니라)에서만 수정되는 변수.

coercion (코어션) 같은 형의 두 인자를 수반하는 연산이 일어나는 동안, 한 형의 인스턴스를 다른 형으로 묵시적으로 변환하는 것. 예를 들어, `int(3.15)`는 실수를 정수 3으로 변환합니다. 하지만, `3+4.5`에서, 각 인자는 다른 형이고 (하나는 `int`, 다른 하나는 `float`), 둘을 더하기 전에 같은 형으로 변환해야 합니다. 그렇지 않으면 `TypeError`를 일으킵니다. 코어션 없이는, 호환되는 형들조차도 프로그래머가 같은 형으로 정규화해주어야 합니다, 예를 들어, 그냥 `3+4.5` 하는 대신 `float(3)+4.5`.

complex number (복소수) 익숙한 실수 시스템의 확장인데, 모든 숫자가 실수부와 허수부의 합으로 표현됩니다. 허수부는 실수에 허수 단위 (-1 의 제곱근)를 곱한 것인데, 종종 수학에서는 i 로, 공학에서는 j 로 표기합니다. 파이썬은 후자의 표기법을 쓰는 복소수를 기본 지원합니다; 허수부는 j 접미사를 붙여서 표기합니다, 예를 들어, `3+1j`. `math` 모듈의 복소수 버전이 필요하다면, `cmath`를 사용합니다. 복소수의 활용은 꽤 수준 높은 수학적 기능입니다. 필요하다고 느끼지 못한다면, 거의 확실히 무시해도 좋습니다.

context manager (컨텍스트 관리자) `__enter__()`와 `__exit__()` 메서드를 정의함으로써 `with` 문에서 보이는 환경을 제어하는 객체. **PEP 343**으로 도입되었습니다.

context variable (컨텍스트 변수) 컨텍스트에 따라 다른 값을 가질 수 있는 변수. 이는 각 실행 스레드가 변수에 대해 다른 값을 가질 수 있는 스레드-로컬 저장소와 비슷합니다. 그러나, 컨텍스트 변수를 통해, 하나의 실행 스레드에 여러 컨텍스트가 있을 수 있으며 컨텍스트 변수의 주 용도는 동시성 비동기 태스크에서 변수를 추적하는 것입니다. `contextvars`를 참조하십시오.

contiguous (연속) 버퍼는 정확히 C-연속 (*C-contiguous*)이거나 포트란 연속 (*Fortran contiguous*)일 때 연속이라고 여겨집니다. 영차원 버퍼는 C-연속이면서 포트란 연속입니다. 일차원 배열에서, 항목들은 서로에 인접하고, 0에서 시작하는 오름차순 인덱스의 순서대로 메모리에 배치되어야 합니다. 다차원 C-연속 배열에서, 메모리 주소의 순서대로 항목들을 방문할 때 마지막 인덱스가 가장 빨리 변합니다. 하지만, 포트란 연속 배열에서는, 첫 번째 인덱스가 가장 빨리 변합니다.

coroutine (코루틴) 코루틴은 서브루틴의 더 일반화된 형태입니다. 서브루틴은 한 지점에서 진입하고 다른 지점에서 탈출합니다. 코루틴은 여러 다른 지점에서 진입하고, 탈출하고, 재개할 수 있습니다. 이것들은 `async def` 문으로 구현할 수 있습니다. **PEP 492**를 보세요.

coroutine function (코루틴 함수) 코루틴 객체를 돌려주는 함수. 코루틴 함수는 `async def` 문으로 정의될 수 있고, `await`와 `async for`와 `async with` 키워드를 포함할 수 있습니다. 이것들은 **PEP 492**에 의해 도입되었습니다.

CPython 파이썬 프로그래밍 언어의 규범적인 구현인데, python.org에서 배포됩니다. 이 구현을 Jython 이나 IronPython 과 같은 다른 것들과 구별할 필요가 있을 때 용어 “CPython”이 사용됩니다.

decorator (데코레이터) 다른 함수를 돌려주는 함수인데, 보통 `@wrapper` 문법을 사용한 함수 변환으로 적용됩니다. 데코레이터의 흔한 예는 `classmethod()`과 `staticmethod()`입니다.

데코레이터 문법은 단지 편의 문법일 뿐입니다. 다음 두 함수 정의는 의미상으로 동등합니다:

```
def f(...):
    ...
f = staticmethod(f)

@staticmethod
def f(...):
    ...
```

같은 개념이 클래스에도 존재하지만, 덜 자주 쓰입니다. 데코레이터에 대한 더 자세한 내용은 함수 정의와 클래스 정의의 설명서를 보면 됩니다.

descriptor (디스크립터) 메서드 `__get__()` 이나 `__set__()` 이나 `__delete__()` 를 정의하는 객체. 클래스 어트리뷰트가 디스크립터일 때, 어트리뷰트 조회는 특별한 연결 작용을 일으킵니다. 보통, $a.b$ 를 읽거나, 쓰거나, 삭제하는데 사용할 때, a 의 클래스 디렉터리에서 b 라고 이름 붙여진 객체를 찾습니다. 하지만 b 가 디스크립터면, 해당하는 디스크립터 메서드가 호출됩니다. 디스크립터를 이해하는 것은 파이썬에 대한 깊은 이해의 열쇠인데, 함수, 메서드, 프로퍼티, 클래스 메서드, 스태틱 메서드, 슈퍼 클래스 참조 등의 많은 기능의 기초를 이루고 있기 때문입니다.

디스크립터의 메서드들에 대한 자세한 내용은 `descriptors` 에 나옵니다.

dictionary (딕셔너리) 임의의 키를 값에 대응시키는 연관 배열 (associative array). 키는 `__hash__()` 와 `__eq__()` 메서드를 갖는 모든 객체가 될 수 있습니다. 필에서 해시라고 부릅니다.

dictionary comprehension A compact way to process all or part of the elements in an iterable and return a dictionary with the results. `results = {n: n ** 2 for n in range(10)}` generates a dictionary containing key n mapped to value $n ** 2$. See comprehensions.

dictionary view (딕셔너리 뷰) `dict.keys()`, `dict.values()`, `dict.items()` 메서드가 돌려주는 객체들을 딕셔너리 뷰라고 부릅니다. 이것들은 딕셔너리 항목들에 대한 동적인 뷰를 제공하는데, 딕셔너리가 변경될 때, 뷰가 이 변화를 반영한다는 뜻입니다. 딕셔너리 뷰를 완전한 리스트로 바꾸려면 `list(dictview)` 를 사용하면 됩니다. `dict-views` 를 보세요.

docstring (독스트링) 클래스, 함수, 모듈에서 첫 번째 표현식으로 나타나는 문자열 리터럴. 스위트가 실행될 때는 무시되지만, 컴파일러에 의해 인지되어 둘러싼 클래스, 함수, 모듈의 `__doc__` 어트리뷰트로 삽입됩니다. 인트로스펙션을 통해 사용할 수 있으므로, 객체의 설명서를 위한 규범적인 장소입니다.

duck-typing (덕 타이핑) 올바른 인터페이스를 가졌는지 판단하는데 객체의 형을 보지 않는 프로그래밍 스타일; 대신, 단순히 메서드나 어트리뷰트가 호출되거나 사용됩니다 (“오리처럼 보이고 오리처럼 꺾꽂히면, 그것은 오리다.”) 특정한 형 대신에 인터페이스를 강조함으로써, 잘 설계된 코드는 다형적인 치환을 허락함으로써 유연성을 개선할 수 있습니다. 덕 타이핑은 `type()` 이나 `isinstance()` 을 사용한 검사를 피합니다. (하지만, 덕 타이핑이 추상 베이스 클래스로 보완될 수 있음에 유의해야 합니다.) 대신에, `hasattr()` 검사나 **EAFP** 프로그래밍을 씁니다.

EAFP 허락보다는 용서를 구하기가 쉽다 (Easier to ask for forgiveness than permission). 이 흔히 볼 수 있는 파이썬 코딩 스타일은, 올바른 키나 어트리뷰트의 존재를 가정하고, 그 가정이 틀리면 예외를 잡습니다. 이 깔끔하고 빠른 스타일은 많은 `try`와 `except` 문의 존재로 특징지어집니다. 이 테크닉은 C와 같은 다른 많은 언어에서 자주 사용되는 **LBYL** 스타일과 대비됩니다.

expression (표현식) 어떤 값으로 구해질 수 있는 문법적인 조각. 다른 말로 표현하면, 표현식은 리터럴, 이름, 어트리뷰트 액세스, 연산자, 함수들과 같은 값을 돌려주는 표현 요소들을 쌓아 올린 것입니다. 다른 많은 언어와 대조적으로, 모든 언어 구성물들이 표현식인 것은 아닙니다. `while` 처럼, 표현식으로 사용할 수 없는 문장들이 있습니다. 대입 또한 문장이고, 표현식이 아닙니다.

extension module (확장 모듈) C 나 C++로 작성된 모듈인데, 파이썬의 C API를 사용해서 핵심이나 사용자 코드와 상호 작용합니다.

f-string (f-문자열) 'f' 나 'F' 를 앞에 붙인 문자열 리터럴들을 흔히 “f-문자열”이라고 부르는데, 포맷 문자열 리터럴의 줄임말입니다. **PEP 498** 을 보세요.

file object (파일 객체) 하부 자원에 대해 파일 지향적 API(`read()` 나 `write()` 같은 메서드들)를 드러내는 객체. 만들어진 방법에 따라, 파일 객체는 실제 디스크 상의 파일이나 다른 저장 장치나 통신 장치(예를 들어, 표준 입출력, 인-메모리 버퍼, 소켓, 파이프, 등등)에 대한 액세스를 중계할 수 있습니다. 파일 객체는 파일류 객체 (*file-like objects*)나 스트림 (*streams*) 이라고도 불립니다.

실제로는 세 부류의 파일 객체들이 있습니다. **날(raw) 바이너리 파일**, **버퍼드(buffered) 바이너리 파일**, **텍스트 파일**. 이들의 인터페이스는 `io` 모듈에서 정의됩니다. 파일 객체를 만드는 규범적인 방법은 `open()` 함수를 쓰는 것입니다.

file-like object (파일류 객체) 파일 객체의 비슷한 말.

finder (파인더) 임포트될 모듈을 위한 로더를 찾으려고 시도하는 객체.

파이썬 3.3. 이후로, 두 종류의 파인더가 있습니다: `sys.meta_path`와 함께 사용하는 **메타 경로 파인더**와 `sys.path_hooks`와 함께 사용하는 **경로 엔트리 파인더**.

더 자세한 내용은 **PEP 302**, **PEP 420**, **PEP 451**에 나옵니다.

floor division (정수 나눗셈) 가장 가까운 정수로 내림하는 수학적 나눗셈. 정수 나눗셈 연산자는 `//` 다. 예를 들어, 표현식 `11 // 4`의 값은 2가 되지만, 실수 나눗셈은 2.75를 돌려줍니다. `(-11) // 4`가 -2.75를 내림한 -3이 됨에 유의해야 합니다. **PEP 238**을 보세요.

function (함수) 호출자에게 어떤 값을 돌려주는 일련의 문장들. 없거나 그 이상의 **인자**가 전달될 수 있는데, 바디의 실행에 사용될 수 있습니다. **매개변수**와 **메서드**와 **function** 섹션도 보세요.

function annotation (함수 어노테이션) 함수 매개변수나 반환 값의 **어노테이션**.

함수 어노테이션은 일반적으로 **형 힌트**로 사용됩니다: 예를 들어, 이 함수는 두 개의 `int` 인자를 받아 들일 것으로 기대되고, 동시에 `int` 반환 값을 줄 것으로 기대됩니다:

```
def sum_two_numbers(a: int, b: int) -> int:
    return a + b
```

함수 어노테이션 문법은 **function** 절에서 설명합니다.

이 기능을 설명하는 **변수 어노테이션**과 **PEP 484**를 참조하세요.

__future__ 프로그래머가 현재 인터프리터와 호환되지 않는 새 언어 기능들을 활성화할 수 있도록 하는 가상 모듈.

`__future__` 모듈을 임포트하고 그 변수들의 값들을 구해서, 새 기능이 언제 처음으로 언어에 추가되었고, 언제부터 그것이 기본이 되는지 볼 수 있습니다:

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

garbage collection (가비지 수거) 더 사용되지 않는 메모리를 반납하는 절차. 파이썬은 참조 횟수 추적과 참조 순환을 감지하고 끊을 수 있는 순환 가비지 수거기를 통해 가비지 수거를 수행합니다. 가비지 수거기는 `gc` 모듈을 사용해서 제어할 수 있습니다.

generator (제너레이터) **제너레이터 이터레이터**를 돌려주는 함수. 일반 함수처럼 보이는데, 일련의 값들을 만드는 `yield` 표현식을 포함한다는 점이 다릅니다. 이 값들은 `for`-루프로 사용하거나 `next()` 함수로 한 번에 하나씩 꺼낼 수 있습니다.

보통 제너레이터 함수를 가리키지만, 어떤 문맥에서는 제너레이터 이터레이터를 가리킵니다. 의도하는 의미가 명확하지 않은 경우는, 완전한 용어를 써서 모호함을 없앱니다.

generator iterator (제너레이터 이터레이터) 제너레이터 함수가 만드는 객체.

각 `yield`는 일시적으로 처리를 중단하고, 그 위치의 (지역 변수들과 대기 중인 `try`-문들을 포함하는) 실행 상태를 기억합니다. 제너레이터 이터레이터가 재개되면, 떠난 곳으로 복귀합니다 (호출마다 새로 시작하는 함수와 대비됩니다).

generator expression (제너레이터 표현식) 이터레이터를 돌려주는 표현식. 루프 변수와 범위를 정의하는 `for` 절과 생략 가능한 `if` 절이 뒤에 붙는 일반 표현식 처럼 보입니다. 결합한 표현식은 둘러싼 함수를 위한 값들을 만들어냅니다:

```
>>> sum(i*i for i in range(10))      # sum of squares 0, 1, 4, ... 81
285
```

generic function (제네릭 함수) 같은 연산을 서로 다른 형들에 대해 구현한 여러 함수로 구성된 함수. 호출 때 어떤 구현이 사용될지는 디스패치 알고리즘에 의해 결정됩니다.

싱글 디스패치 용어집 항목과 `functools singledispatch()` 데코레이터와 **PEP 443**도 보세요.

GIL 전역 인터프리터 록 을 보세요.

global interpreter lock (전역 인터프리터 록) 한 번에 오직 하나의 스레드가 파이썬 바이트 코드를 실행하도록 보장하기 위해 CPython 인터프리터가 사용하는 메커니즘. (dict와 같은 중요한 내장형들을 포함하는) 객체 모델이 묵시적으로 동시 액세스에 대해 안전하도록 만들어서 CPython 구현을 단순하게 만듭니다. 인터프리터 전체를 잠그는 것은 인터프리터를 다중스레드화하기 쉽게 만드는 대신, 다중 프로세서 기계가 제공하는 병렬성의 많은 부분을 희생합니다.

하지만, 어떤 확장 모듈들은, 표준이나 제삼자 모두, 압축이나 해싱 같은 계산 집약적인 작업을 수행할 때는 GIL을 반납하도록 설계되었습니다. 또한, I/O를 할 때는 항상 GIL을 반납합니다.

(훨씬 더 미세하게 공유 데이터를 잠그는) “스레드에 자유로운(free-threaded)” 인터프리터를 만들고자 하는 과거의 노력은 성공적이지 못했는데, 혼란 단일 프로세서 경우의 성능 저하가 심하기 때문입니다. 이 성능 이슈를 극복하는 것은 구현을 훨씬 복잡하게 만들어서 유지 비용이 더 들어갈 것으로 여겨지고 있습니다.

hash-based pyc (해시 기반 pyc) 유효성을 판별하기 위해 해당 소스 파일의 최종 수정 시간이 아닌 해시를 사용하는 바이트 코드 캐시 파일. `pyc-invalidation`을 참조하세요.

hashable (해시 가능) 객체가 일생 그 값이 변하지 않는 해시값을 갖고 (`__hash__()` 메서드가 필요합니다), 다른 객체와 비교될 수 있으면 (`__eq__()` 메서드가 필요합니다), 해시 가능하다고 합니다. 같다고 비교되는 해시 가능한 객체들의 해시값은 같아야 합니다.

해시 가능성은 객체를 딕셔너리의 키나 집합의 멤버로 사용할 수 있게 하는데, 이 자료 구조들이 내부적으로 해시값을 사용하기 때문입니다.

대부분 파이썬의 불변 내장 객체들은 해시 가능합니다; (리스트나 딕셔너리 같은) 가변 컨테이너들은 그렇지 않습니다; (튜플이나 `frozenset` 같은) 불변 컨테이너들은 그들의 요소들이 해시 가능할 때만 해시 가능합니다. 사용자 정의 클래스의 인스턴스 객체들은 기본적으로 해시 가능합니다. (자기 자신을 제외하고는) 모두 다르다고 비교되고, 해시값은 `id()` 로 부터 만들어집니다.

IDLE 파이썬을 위한 통합 개발 환경 (Integrated Development Environment). IDLE은 파이썬의 표준 배포판에 따라오는 기초적인 편집기와 인터프리터 환경입니다.

immutable (불변) 고정된 값을 갖는 객체. 불변 객체는 숫자, 문자열, 튜플을 포함합니다. 이런 객체들은 변경될 수 없습니다. 새 값을 저장하려면 새 객체를 만들어야 합니다. 변하지 않는 해시값이 있어야 하는 곳에서 중요한 역할을 합니다, 예를 들어, 딕셔너리의 키.

import path (임포트 경로) 경로 기반 파인더가 임포트 할 모듈을 찾기 위해 검색하는 장소들 (또는 경로 엔트리)의 목록. 임포트 하는 동안, 이 장소들의 목록은 보통 `sys.path`로부터 옵니다, 하지만 서브 패키지의 경우 부모 패키지의 `__path__` 어트리뷰트로부터 올 수도 있습니다.

importing (임포트) 한 모듈의 파이썬 코드가 다른 모듈의 파이썬 코드에서 사용될 수 있도록 하는 절차.

importer (임porter) 모듈을 찾기도 하고 로드 하기도 하는 객체; 동시에 파인더 이자 로더 객체입니다.

interactive (대화형) 파이썬은 대화형 인터프리터를 갖고 있는데, 인터프리터 프롬프트에서 문장과 표현식을 입력할 수 있고, 즉각 실행된 결과를 볼 수 있다는 뜻입니다. 인자 없이 단지 `python`을 실행하세요 (컴퓨터의 주메뉴에서 선택하는 것도 가능할 수 있습니다). 새 아이디어를 검사하거나 모듈과 패키지를 들여다보는 매우 강력한 방법입니다 (`help(x)`를 기억하세요).

interpreted (인터프리티드) 바이트 코드 컴파일러의 존재 때문에 그 부분이 흐릿해지기는 하지만, 파이썬은 컴파일 언어가 아니라 인터프리터 언어입니다. 이것은 명시적으로 실행 파일을 만들지 않고도, 소스 파일을 직접 실행할 수 있다는 뜻입니다. 그 프로그램이 좀 더 천천히 실행되기는 하지만, 인터프리터 언어는 보통 컴파일 언어보다 짧은 개발/디버깅 주기를 갖습니다. [대화형](#)도 보세요.

interpreter shutdown (인터프리터 종료) 종료하라는 요청을 받을 때, 파이썬 인터프리터는 특별한 시기에 진입하는데, 모듈이나 여러 가지 중요한 내부 구조들과 같은 모든 할당된 자원들을 단계적으로 반납합니다. 또한, [가비지 수거기](#)를 여러 번 호출합니다. 사용자 정의 파괴자나 `weakref` 콜백에 있는 코드들의 실행을 시작시킬 수 있습니다. 종료 시기 동안 실행되는 코드는 다양한 예외들을 만날 수 있는데, 그것이 의존하는 자원들이 더 기능하지 않을 수 있기 때문입니다(흔한 예는 라이브러리 모듈이나 경고 장치들입니다).

인터프리터 종료의 주된 원인은 실행되는 `__main__` 모듈이나 스크립트가 실행을 끝내는 것입니다.

iterable (이터러블) 멤버들을 한 번에 하나씩 돌려줄 수 있는 객체. 이터러블의 예로는 모든 (`list`, `str`, `tuple` 같은) 시퀀스 형들, `dict` 같은 몇몇 비 시퀀스 형들, [파일 객체들](#), `__iter__()` 나 [시퀀스](#) 개념을 구현하는 `__getitem__()` 메서드를 써서 정의한 모든 클래스의 객체들이 있습니다.

이터러블은 `for` 루프에 사용될 수 있고, 시퀀스를 필요로 하는 다른 많은 곳 (`zip()`, `map()`, ...) 에 사용될 수 있습니다. 이터러블 객체가 내장 함수 `iter()` 에 인자로 전달되면, 그 객체의 이터레이터를 돌려줍니다. 이 이터레이터는 값들의 집합을 한 번 거치는 동안 유효합니다. 이터러블을 사용할 때, 보통은 `iter()` 를 호출하거나, 이터레이터 객체를 직접 다룰 필요는 없습니다. `for` 문은 이것들을 여러 분을 대신해서 자동으로 해주는데, 루프를 도는 동안 이터레이터를 잡아둘 이름 없는 변수를 만듭니다. [이터레이터](#), [시퀀스](#), [제너레이터](#)도 보세요.

iterator (이터레이터) 데이터의 스트림을 표현하는 객체. 이터레이터의 `__next__()` 메서드를 반복적으로 호출하면 (또는 내장 함수 `next()` 로 전달하면) 스트림에 있는 항목들을 차례대로 돌려줍니다. 더 이상의 데이터가 없을 때는 대신 `StopIteration` 예외를 일으킵니다. 이 지점에서, 이터레이터 객체는 소진되고, 이후의 모든 `__next__()` 메서드 호출은 `StopIteration` 예외를 다시 일으키기만 합니다. 이터레이터는 이터레이터 객체 자신을 돌려주는 `__iter__()` 메서드를 가질 것이 요구되기 때문에, 이터레이터는 이터러블이기도 하고 다른 이터러블들을 받아들이는 대부분의 곳에서 사용될 수 있습니다. 중요한 예외는 여러 번의 이터레이션을 시도하는 코드입니다. (`list` 같은) 컨테이너 객체는 `iter()` 함수로 전달하거나 `for` 루프에 사용할 때마다 새 이터레이터를 만듭니다. 이런 것을 이터레이터에 대해서 수행하려고 하면, 지난 이터레이션에 사용된 이미 소진된 이터레이터를 돌려줘서, 빈 컨테이너처럼 보이게 만듭니다.

`typeiter` 에 더 자세한 내용이 있습니다.

key function (키 함수) 키 함수 또는 콜레이션(`collation`) 함수는 정렬(`sorting`)이나 배열(`ordering`)에 사용되는 값을 돌려주는 콜러블입니다. 예를 들어, `locale.strxfrm()` 은 로케일 특정 방식을 따르는 정렬 키를 만드는 데 사용됩니다.

파이썬의 많은 도구가 요소들이 어떻게 순서 지어지고 묶이는지를 제어하기 위해 키 함수를 받아들입니다. 이런 것들에는 `min()`, `max()`, `sorted()`, `list.sort()`, `heapq.merge()`, `heapq.nsmallest()`, `heapq.nlargest()`, `itertools.groupby()` 이 있습니다.

키 함수를 만드는 데는 여러 방법이 있습니다. 예를 들어, `str.lower()` 메서드는 케이스 구분 없는 정렬을 위한 키 함수로 사용될 수 있습니다. 대안적으로, 키 함수는 `lambda` 표현식으로 만들 수도 있는데, 이런 식입니다: `lambda r: (r[0], r[2])`. 또한, `operator` 모듈은 세 개의 키 함수 생성자를 제공합니다: `attrgetter()`, `itemgetter()`, `methodcaller()`. 키 함수를 만들고 사용하는 법에 대한 예로 [Sorting HOW TO](#) 를 보세요.

keyword argument (키워드 인자) [인자](#)를 보세요.

lambda (람다) 호출될 때 값이 구해지는 하나의 [표현식](#)으로 구성된 이름 없는 인라인 함수. 람다 함수를 만드는 문법은 `lambda [parameters]: expression` 입니다.

LBYL 뛰기 전에 보라 (Look before you leap). 이 코딩 스타일은 호출이나 조회를 하기 전에 명시적으로 사전 조건들을 검사합니다. 이 스타일은 [EAFP](#) 접근법과 대비되고, 많은 `if` 문의 존재로 특징지어집니다.

다중 스레드 환경에서, LBYL 접근법은 “보기”와 “뛰기” 간에 경쟁 조건을 만들게 될 위험이 있습니다.

예를 들어, 코드 `if key in mapping: return mapping[key]` 는 검사 후에, 하지만 조회 전에, 다른 스레드가 `key`를 `mapping`에서 제거하면 실패할 수 있습니다. 이런 이슈는 록이나 EAFP 접근법을 사용함으로써 해결될 수 있습니다.

list (리스트) 내장 파이썬 시퀀스. 그 이름에도 불구하고, 원소에 대한 액세스가 $O(1)$ 이기 때문에, 연결 리스트(linked list)보다는 다른 언어의 배열과 유사합니다.

list comprehension (리스트 컴프리헨션) 시퀀스의 요소들 전부 또는 일부를 처리하고 그 결과를 리스트로 돌려주는 간결한 방법. `result = ['{:04x}'.format(x) for x in range(256) if x % 2 == 0]` 는 0에서 255 사이에 있는 짝수들의 16진수 (0x..) 들을 포함하는 문자열의 리스트를 만듭니다. `if` 절은 생략할 수 있습니다. 생략하면, `range(256)`에 있는 모든 요소가 처리됩니다.

loader (로더) 모듈을 로드하는 객체. `load_module()`이라는 이름의 메서드를 정의해야 합니다. 로더는 보통 **파인더**가 돌려줍니다. 자세한 내용은 [PEP 302](#)를, 추상 베이스 클래스는 `importlib.abc.Loader`를 보세요.

magic method (매직 메서드) 특수 메서드 의 비공식적인 비슷한 말.

mapping (매핑) 임의의 키 조회를 지원하고 `Mapping`이나 `MutableMapping` 추상 베이스 클래스에 지정된 메서드들을 구현하는 컨테이너 객체. 예로는 `dict`, `collections.defaultdict`, `collections.OrderedDict`, `collections.Counter`를 들 수 있습니다.

meta path finder (메타 경로 파인더) `sys.meta_path`의 검색이 돌려주는 **파인더**. 메타 경로 파인더는 **경로 엔트리 파인더**와 관련되어 있기는 하지만 다릅니다.

메타 경로 파인더가 구현하는 메서드들에 대해서는 `importlib.abc.MetaPathFinder`를 보면 됩니다.

metaclass (메타 클래스) 클래스의 클래스. 클래스 정의는 클래스 이름, 클래스 디렉터리, 베이스 클래스들의 목록을 만듭니다. 메타 클래스는 이 세 인자를 받아서 클래스를 만드는 책임을 집니다. 대부분의 객체 지향형 프로그래밍 언어들은 기본 구현을 제공합니다. 파이썬을 특별하게 만드는 것은 커스텀 메타 클래스를 만들 수 있다는 것입니다. 대부분 사용자에게는 이 도구가 전혀 필요 없지만, 필요가 생길 때, 메타 클래스는 강력하고 우아한 해법을 제공합니다. 어트리뷰트 액세스의 로깅(logging), 스레드 안전성의 추가, 객체 생성 추적, 싱글톤 구현과 많은 다른 작업에 사용됐습니다.

metaclasses에서 더 자세한 내용을 찾을 수 있습니다.

method (메서드) 클래스 바디 안에서 정의되는 함수. 그 클래스의 인스턴스의 어트리뷰트로서 호출되면, 그 메서드는 첫 번째 **인자**(보통 `self`라고 불린다)로 인스턴스 객체를 받습니다. **함수**와 **중첩된 스코프**를 보세요.

method resolution order (메서드 결정 순서) 메서드 결정 순서는 조회하는 동안 멤버를 검색하는 베이스 클래스들의 순서입니다. 2.3 릴리스부터 파이썬 인터프리터에 사용된 알고리즘의 상세한 내용은 [The Python 2.3 Method Resolution Order](#)를 보면 됩니다.

module (모듈) 파이썬 코드의 조직화 단위를 담당하는 객체. 모듈은 임의의 파이썬 객체들을 담는 이름 공간을 갖습니다. 모듈은 **임포트** 절차에 의해 파이썬으로 로드됩니다.

패키지도 보세요.

module spec (모듈 스펙) 모듈을 로드하는데 사용되는 임포트 관련 정보들을 담고 있는 이름 공간. `importlib.machinery.ModuleSpec`의 인스턴스.

MRO 메서드 결정 순서를 보세요.

mutable (가변) 가변 객체는 값이 변할 수 있지만 `id()`는 일정하게 유지합니다. **불변**도 보세요.

named tuple (네임드 튜플) “named tuple(네임드 튜플)”이라는 용어는 튜플에서 상속하고 이름 붙은 어트리뷰트를 사용하여 인덱스 할 수 있는 요소에 액세스 할 수 있는 모든 형이나 클래스에 적용됩니다. 형이나 클래스에는 다른 기능도 있을 수 있습니다.

`time.localtime()`과 `os.stat()`가 반환한 값을 포함하여, 여러 내장형이 네임드 튜플입니다. 또 다른 예는 `sys.float_info`입니다:


```

>>> 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

```

일부 네임드 튜플은 내장형(위의 예)입니다. 또는, tuple에서 상속하고 이름 붙은 필드를 정의하는 일반 클래스 정의로 네임드 튜플을 만들 수 있습니다. 이러한 클래스는 직접 작성하거나 팩토리 함수 `collections.namedtuple()`로 만들 수 있습니다. 후자의 기법은 직접 작성하거나 내장 네임드 튜플에서는 찾을 수 없는 몇 가지 추가 메서드를 추가하기도 합니다.

namespace (이름 공간) 변수가 저장되는 장소. 이름 공간은 딕셔너리로 구현됩니다. 객체에 중첩된 이름 공간(메서드에서) 뿐만 아니라 지역, 전역, 내장 이름 공간이 있습니다. 이름 공간은 이름 충돌을 방지해서 모듈성을 지원합니다. 예를 들어, 함수 `builtins.open`과 `os.open()`은 그들의 이름 공간에 의해 구별됩니다. 또한, 이름 공간은 어떤 모듈이 함수를 구현하는지를 분명하게 만들어서 가독성과 유지 보수성에 도움을 줍니다. 예를 들어, `random.seed()` 또는 `itertools.islice()`라고 쓰면 그 함수들이 각각 `random`과 `itertools` 모듈에 의해 구현되었음이 명확해집니다.

namespace package (이름 공간 패키지) 오직 서브 패키지들의 컨테이너로만 기능하는 **PEP 420 패키지**. 이름 공간 패키지는 물리적인 실체가 없을 수도 있고, 특히 `__init__.py` 파일이 없으므로 **정규 패키지**와는 다릅니다.

모듈도 보세요.

nested scope (중첩된 스코프) 둘러싼 정의에서 변수를 참조하는 능력. 예를 들어, 다른 함수 내부에서 정의된 함수는 바깥 함수에 있는 변수들을 참조할 수 있습니다. 중첩된 스코프는 기본적으로는 참조만 가능할 뿐, 대입은 되지 않는다는 것에 주의해야 합니다. 지역 변수들은 가장 내부의 스코프에서 읽고 씁니다. 마찬가지로, 전역 변수들은 전역 이름 공간에서 읽고 씁니다. `nonlocal`은 바깥 스코프에 쓰는 것을 허락합니다.

new-style class (뉴스타일 클래스) 지금은 모든 클래스 객체에 사용되고 있는 클래스 버전의 예전 이름. 초기의 파이썬 버전에서는, 오직 뉴스타일 클래스만 `__slots__`, 디스크립터, 프라퍼티, `__getattr__()`, 클래스 메서드, 스태틱 메서드와 같은 파이썬의 새롭고 다양한 기능들을 사용할 수 있었습니다.

object (객체) 상태(어트리뷰트나 값)를 갖고 동작(메서드)이 정의된 모든 데이터. 또한, 모든 뉴스타일 클래스의 최종적인 베이스 클래스입니다.

package (패키지) 서브 모듈들이나, 재귀적으로 서브 패키지들을 포함할 수 있는 파이썬 **모듈**. 기술적으로, 패키지는 `__path__` 어트리뷰트가 있는 파이썬 모듈입니다.

정규 패키지 와 이름 공간 패키지 도 보세요.

parameter (매개변수) 함수(또는 메서드) 정의에서 함수가 받을 수 있는 **인자**(또는 어떤 경우 인자들)를 지정하는 이름 붙은 엔티티. 다섯 종류의 매개변수가 있습니다:

- 위치-키워드 (*positional-or-keyword*): 위치 인자 나 키워드 인자로 전달될 수 있는 인자를 지정합니다. 이것이 기본 형태의 매개변수입니다, 예를 들어 다음에서 `foo`와 `bar`:

```
def func(foo, bar=None): ...
```

- 위치-전용 (*positional-only*): 위치로만 제공될 수 있는 인자를 지정합니다. 위치 전용 매개변수는 함수 정의의 매개변수 목록에 / 문자를 포함하고 그 뒤에 정의할 수 있습니다, 예를 들어 다음에서 `posonly1`과 `posonly2`:

```
def func(posonly1, posonly2, /, positional_or_keyword): ...
```

- 키워드-전용 (*keyword-only*): 키워드로만 제공될 수 있는 인자를 지정합니다. 키워드-전용 매개변수는 함수 정의의 매개변수 목록에서 앞에 하나의 가변-위치 매개변수나 *를 그대로 포함해서 정의할 수 있습니다. 예를 들어, 다음에서 *kw_only1* 와 *kw_only2*:

```
def func(arg, *, kw_only1, kw_only2): ...
```

- 가변-위치 (*var-positional*): (다른 매개변수들에 의해서 이미 받아들여진 위치 인자들에 더해) 제공될 수 있는 위치 인자들의 임의의 시퀀스를 지정합니다. 이런 매개변수는 매개변수 이름에 *를 앞에 붙여서 정의될 수 있습니다, 예를 들어 다음에서 *args*:

```
def func(*args, **kwargs): ...
```

- 가변-키워드 (*var-keyword*): (다른 매개변수들에 의해서 이미 받아들여진 키워드 인자들에 더해) 제공될 수 있는 임의의 개수 키워드 인자들을 지정합니다. 이런 매개변수는 매개변수 이름에 **를 앞에 붙여서 정의될 수 있습니다, 예를 들어 위의 예에서 *kwargs*.

매개변수는 선택적 인자들을 위한 기본값뿐만 아니라 선택적이거나 필수 인자들을 지정할 수 있습니다.

인자 용어집 항목, 인자와 매개변수의 차이에 나오는 FAQ 질문, `inspect.Parameter` 클래스, `function` 절, [PEP 362](#)도 보세요.

path entry (경로 엔트리) 경로 기반 파인더가 임포트 할 모듈들을 찾기 위해 참고하는 임포트 경로 상의 하나의 장소.

path entry finder (경로 엔트리 파인더) `sys.path_hooks`에 있는 콜러블 (즉, [경로 엔트리](#) [혹](#)) 이 돌려주는 파인더 인데, 주어진 [경로 엔트리](#)로 모듈을 찾는 방법을 알고 있습니다.

경로 엔트리 파인더들이 구현하는 메서드들은 `importlib.abc.PathEntryFinder`에 나옵니다.

path entry hook (경로 엔트리 [혹](#)) `sys.path_hook` 리스트에 있는 콜러블인데, 특정 [경로 엔트리](#)에서 모듈을 찾는 법을 알고 있다면 [경로 엔트리 파인더](#)를 돌려줍니다.

path based finder (경로 기반 파인더) 기본 메타 경로 파인더들 중 하나인데, [임포트 경로](#)에서 모듈을 찾습니다.

path-like object (경로류 객체) 파일 시스템 경로를 나타내는 객체. 경로류 객체는 경로를 나타내는 `str` 나 `bytes` 객체이거나 `os.PathLike` 프로토콜을 구현하는 객체입니다. `os.PathLike` 프로토콜을 지원하는 객체는 `os.fspath()` 함수를 호출해서 `str` 나 `bytes` 파일 시스템 경로로 변환될 수 있습니다; 대신 `os.fsdecode()` 와 `os.fsencode()` 는 각각 `str` 나 `bytes` 결과를 보장하는데 사용될 수 있습니다. [PEP 519](#)로 도입되었습니다.

PEP 파이썬 개선 제안. PEP는 파이썬 커뮤니티에 정보를 제공하거나 파이썬 또는 그 프로세스 또는 환경에 대한 새로운 기능을 설명하는 설계 문서입니다. PEP는 제안된 기능에 대한 간결한 기술 사양 및 근거를 제공해야 합니다.

PEP는 주요 새로운 기능을 제안하고 문제에 대한 커뮤니티 입력을 수집하며 파이썬에 들어간 설계 결정을 문서로 만들기 위한 기본 메커니즘입니다. PEP 작성자는 커뮤니티 내에서 합의를 구축하고 반대 의견을 문서화 할 책임이 있습니다.

[PEP 1](#) 참조하세요.

portion (포션) [PEP 420](#)에서 정의한 것처럼, 이름 공간 패키지에 이바지하는 하나의 디렉터리에 들어있는 파일들의 집합 (zip 파일에 저장되는 것도 가능합니다).

positional argument (위치 인자) [인자](#)를 보세요.

provisional API (잠정 API) 잠정 API는 표준 라이브러리의 과거 호환성 보장으로부터 신중히 제외된 것입니다. 인터페이스의 큰 변화가 예상되지는 않지만, 잠정적이라고 표시되는 한, 코어 개발자들이 필요하다고 생각한다면 과거 호환성이 유지되지 않는 변경이 일어날 수 있습니다. 그런 변경은 불필요한 방식으로 일어나지는 않을 것입니다 — API를 포함하기 전에 놓친 중대하고 근본적인 결함이 발견된 경우에만 일어날 것입니다.

잠정 API에서조차도, 과거 호환성이 유지되지 않는 변경은 “최후의 수단”으로 여겨집니다 - 모든 식별된 문제들에 대해 과거 호환성을 유지하는 해법을 찾으려는 모든 시도가 선행됩니다.

이 절차는 표준 라이브러리가 오랜 시간 동안 잘못된 설계 오류에 발목 잡히지 않고 발전할 수 있도록 만듭니다. 더 자세한 내용은 [PEP 411](#)을 보면 됩니다.

provisional package (잠정 패키지) 잠정 API를 보세요.

Python 3000 (파이썬 3000) 파이썬 3.x 배포 라인의 별명 (버전 3의 배포가 먼 미래의 이야기던 시절에 만들어진 이름이다.) 이것을 “Py3k”로 줄여 쓰기도 합니다.

Pythonic (파이썬다운) 다른 언어들에서 일반적인 개념들을 사용해서 코드를 구현하는 대신, 파이썬 언어에서 가장 자주 사용되는 이디엄들을 가까이 따르는 아이디어나 코드 조각. 예를 들어, 파이썬에서 자주 쓰는 이디엄은 `for` 문을 사용해서 이터러블의 모든 요소로 루핑하는 것입니다. 다른 많은 언어에는 이런 종류의 구성물이 없으므로, 파이썬에 익숙하지 않은 사람들은 대신에 숫자 카운터를 사용하기도 합니다:

```
for i in range(len(food)):
    print(food[i])
```

더 깔끔한, 파이썬다운 방법은 이렇습니다:

```
for piece in food:
    print(piece)
```

qualified name (정규화된 이름) 모듈의 전역 스코프에서 모듈에 정의된 클래스, 함수, 메서드에 이르는 “경로”를 보여주는 점으로 구분된 이름. [PEP 3155](#)에서 정의됩니다. 최상위 함수와 클래스의 경우에, 정규화된 이름은 객체의 이름과 같습니다:

```
>>> class C:
...     class D:
...         def meth(self):
...             pass
...
>>> C.__qualname__
'C'
>>> C.D.__qualname__
'C.D'
>>> C.D.meth.__qualname__
'C.D.meth'
```

모듈을 가리키는데 사용될 때, 완전히 정규화된 이름 (*fully qualified name*)은 모든 부모 패키지들을 포함해서 모듈로 가는 점으로 분리된 이름을 의미합니다, 예를 들어, `email.mime.text`:

```
>>> import email.mime.text
>>> email.mime.text.__name__
'email.mime.text'
```

reference count (참조 횟수) 객체에 대한 참조의 개수. 객체의 참조 횟수가 0으로 떨어지면, 메모리가 반납됩니다. 참조 횟수 추적은 일반적으로 파이썬 코드에 노출되지는 않지만, [CPython](#) 구현의 핵심 요소입니다. `sys` 모듈은 특정 객체의 참조 횟수를 돌려주는 `getrefcount()`을 정의합니다.

regular package (정규 패키지) `__init__.py` 파일을 포함하는 디렉터리와 같은 전통적인 패키지.

이름 공간 패키지도 보세요.

__slots__ 클래스 내부의 선언인데, 인스턴스 어트리뷰트들을 위한 공간을 미리 선언하고 인스턴스 디렉터리를 제거함으로써 메모리를 절감하는 효과를 줍니다. 인기 있기는 하지만, 이 테크닉은 올바르게 사용하기가 좀 까다로운 편이라서, 메모리에 민감한 응용 프로그램에서 많은 수의 인스턴스가 있는 특별한 경우로 한정하는 것이 좋습니다.

sequence (시퀀스) `__getitem__()` 특수 메서드를 통해 정수 인덱스를 사용한 빠른 요소 액세스를 지원하고, 시퀀스의 길이를 돌려주는 `__len__()` 메서드를 정의하는 **이터러블**. 몇몇 내장 시퀀스들을 나열해보면, `list`, `str`, `tuple`, `bytes` 가 있습니다. `dict` 또한 `__getitem__()` 과 `__len__()` 을 지원하지만, 조회에 정수 대신 임의의 불변 키를 사용하기 때문에 시퀀스가 아니라 매핑으로 취급된다는 것에 주의해야 합니다.

The `collections.abc.Sequence` abstract base class defines a much richer interface that goes beyond just `__getitem__()` and `__len__()`, adding `count()`, `index()`, `__contains__()`, and `__reversed__()`. Types that implement this expanded interface can be registered explicitly using `register()`.

set comprehension A compact way to process all or part of the elements in an iterable and return a set with the results. `results = {c for c in 'abracadabra' if c not in 'abc'}` generates the set of strings `{'r', 'd'}`. See comprehensions.

single dispatch (싱글 디스패치) 구현이 하나의 인자의 형에 기초해서 결정되는 **제네릭 함수** 디스패치의 한 형태.

slice (슬라이스) 보통 시퀀스의 일부를 포함하는 객체. 슬라이스는 서브 스크립트 표기법을 사용해서 만듭니다. `variable_name[1:3:5]` 처럼, `[]` 안에서 여러 개의 숫자를 콜론으로 분리합니다. 대괄호 (서브 스크립트) 표기법은 내부적으로 slice 객체를 사용합니다.

special method (특수 메서드) 파이썬이 형에 어떤 연산을, 덧셈 같은, 실행할 때 묵시적으로 호출되는 메서드. 이런 메서드는 두 개의 밑줄로 시작하고 끝나는 이름을 갖고 있습니다. 특수 메서드는 `specialnames` 에 문서로 만들어져 있습니다.

statement (문장) 문장은 스위트 (코드의 “블록(block)”) 를 구성하는 부분입니다. 문장은 **표현식** 이거나 키워드를 사용하는 여러 가지 구조물 중의 하나입니다. 가령 `if`, `while`, `for`.

text encoding (텍스트 인코딩) 유니코드 문자열을 바이트열로 인코딩하는 코덱.

text file (텍스트 파일) `str` 객체를 읽고 쓸 수 있는 **파일 객체**. 종종, 텍스트 파일은 실제로는 바이트 지향 데이터 스트림을 액세스하고 **텍스트 인코딩** 을 자동 처리합니다. 텍스트 파일의 예로는 텍스트 모드 ('r' 또는 'w') 로 열린 파일, `sys.stdin`, `sys.stdout`, `io.StringIO` 의 인스턴스를 들 수 있습니다.

바이트열류 객체 를 읽고 쓸 수 있는 파일 객체에 대해서는 **바이너리 파일** 도 참조하세요.

triple-quoted string (삼중 따옴표 된 문자열) 따옴표 (") 나 작은따옴표 (') 세 개로 둘러싸인 문자열. 그냥 따옴표 하나로 둘러싸인 문자열에 없는 기능을 제공하지는 않지만, 여러 가지 이유에서 쓸모가 있습니다. 이스케이프 되지 않은 작은따옴표나 큰따옴표를 문자열 안에 포함할 수 있도록 하고, 연결 문자를 쓰지 않고도 여러 줄에 걸쳐 쓸 수 있는데, 독스트링을 쓸 때 특히 쓸모 있습니다.

type (형) 파이썬 객체의 형은 그것이 어떤 종류의 객체인지를 결정합니다; 모든 객체는 형이 있습니다. 객체의 형은 `__class__` 어트리뷰트로 액세스할 수 있거나 `type(obj)` 로 얻을 수 있습니다.

type alias (형 에일리어스) 형을 식별자에 대입하여 만들어지는 형의 동의어.

형 에일리어스는 **형 힌트**를 단순화하는 데 유용합니다. 예를 들면:

```
from typing import List, Tuple

def remove_gray_shades(
    colors: List[Tuple[int, int, int]]) -> List[Tuple[int, int, int]]:
    pass
```

는 다음과 같이 더 읽기 쉽게 만들 수 있습니다:

```
from typing import List, Tuple

Color = Tuple[int, int, int]
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
def remove_gray_shades(colors: List[Color]) -> List[Color]:
    pass
```

이 기능을 설명하는 typing과 [PEP 484](#)를 참조하세요.

type hint (형 힌트) 변수, 클래스 어트리뷰트 및 함수 매개변수 나 반환 값의 기대되는 형을 지정하는 어노테이션.

형 힌트는 선택 사항이며 파이썬에서 강제되지는 않습니다. 하지만, 정적 형 분석 도구에 유용하며 IDE의 코드 완성 및 리팩토링을 돕습니다.

지역 변수를 제외하고, 전역 변수, 클래스 어트리뷰트 및 함수의 형 힌트는 `typing.get_type_hints()`를 사용하여 액세스할 수 있습니다.

이 기능을 설명하는 typing과 [PEP 484](#)를 참조하세요.

universal newlines (유니버설 줄 넘김) 다음과 같은 것들을 모두 줄의 끝으로 인식하는, 텍스트 스트림을 해석하는 태도: 유닉스 개행 문자 관례 '\n', 윈도우즈 관례 '\r\n', 예전의 매킨토시 관례 '\r'. 추가적인 사용에 관해서는 `bytes.splitlines()` 뿐만 아니라 [PEP 278](#)와 [PEP 3116](#)도 보세요.

variable annotation (변수 어노테이션) 변수 또는 클래스 어트리뷰트의 어노테이션.

변수 또는 클래스 어트리뷰트에 어노테이션을 달 때 대입은 선택 사항입니다:

```
class C:
    field: 'annotation'
```

변수 어노테이션은 일반적으로 형 힌트로 사용됩니다: 예를 들어, 이 변수는 `int` 값을 가질 것으로 기대됩니다:

```
count: int = 0
```

변수 어노테이션 문법은 섹션 [annassign](#)에서 설명합니다.

이 기능을 설명하는 함수 어노테이션, [PEP 484](#) 및 [PEP 526](#)을 참조하세요.

virtual environment (가상 환경) 파이썬 사용자와 응용 프로그램이, 같은 시스템에서 실행되는 다른 파이썬 응용 프로그램들의 동작에 영향을 주지 않으면서, 파이썬 배포 패키지들을 설치하거나 업그레이드하는 것을 가능하게 하는, 협력적으로 격리된 실행 환경.

`venv`도 보세요.

virtual machine (가상 기계) 소프트웨어만으로 정의된 컴퓨터. 파이썬의 가상 기계는 바이트 코드 컴파일러가 출력하는 바이트 코드를 실행합니다.

Zen of Python (파이썬 젠) 파이썬 디자인 원리와 철학들의 목록인데, 언어를 이해하고 사용하는 데 도움이 됩니다. 이 목록은 대화형 프롬프트에서 “`import this`”를 입력하면 보입니다.

이 설명서에 관하여

이 설명서는 `reStructuredText` 소스에서 만들어진 것으로, 파이썬 설명서를 위해 특별히 제작된 문서 처리기인 `Sphinx` 를 사용했습니다.

설명서와 이를 위한 툴체인 개발은 파이썬 자체와 마찬가지로 전적으로 자원봉사자의 노력입니다. 기여하고 싶다면, 참여 방법에 대한 정보는 `reporting-bugs` 페이지를 참고하십시오. 새로운 자원봉사자는 언제나 환영합니다!

다음 분들에게 많은 감사를 드립니다:

- Fred L. Drake, Jr., 원래 파이썬 설명서 도구 집합의 작성자이자 많은 콘텐츠의 작가;
- `reStructuredText`와 `Docutils` 스위트를 만드는 `Docutils` 프로젝트.
- Fredrik Lundh, 그의 `Alternative Python Reference` 프로젝트에서 `Sphinx`가 많은 아이디어를 얻었습니다.

B.1 파이썬 설명서의 공헌자들

많은 사람이 파이썬 언어, 파이썬 표준 라이브러리 및 파이썬 설명서에 기여했습니다. 기여자의 부분적인 목록은 파이썬 소스 배포판의 `Misc/ACKS` 를 참조하십시오.

파이썬이 이런 멋진 설명서를 갖게 된 것은 파이썬 커뮤니티의 입력과 기여 때문입니다 – 감사합니다!

역사와 라이선스

C.1 소프트웨어의 역사

파이썬은 ABC라는 언어의 후계자로서 네덜란드의 Stichting Mathematisch Centrum (CWI, <https://www.cwi.nl/> 참조)의 Guido van Rossum에 의해 1990년대 초반에 만들어졌습니다. 파이썬에는 다른 사람들의 많은 공헌이 포함되었지만, Guido는 파이썬의 주요 저자로 남아 있습니다.

1995년, Guido는 Virginia의 Reston에 있는 Corporation for National Research Initiatives(CNRI, <https://www.cnri.reston.va.us/> 참조)에서 파이썬 작업을 계속했고, 이곳에서 여러 버전의 소프트웨어를 출시했습니다.

2000년 5월, Guido와 파이썬 핵심 개발팀은 BeOpen.com으로 옮겨서 BeOpen PythonLabs 팀을 구성했습니다. 같은 해 10월, PythonLabs 팀은 Digital Creations(현재 Zope Corporation; <https://www.zope.org/> 참조)로 옮겼습니다. 2001년, 파이썬 소프트웨어 재단(PSF, <https://www.python.org/psf/> 참조)이 설립되었습니다. 이 단체는 파이썬 관련 지적 재산을 소유하도록 특별히 설립된 비영리 조직입니다. Zope Corporation은 PSF의 후원 회원입니다.

모든 파이썬 배포판은 공개 소스입니다(공개 소스 정의에 대해서는 <https://opensource.org/>를 참조하십시오). 역사적으로, 대부분(하지만 전부는 아닙니다) 파이썬 배포판은 GPL과 호환됩니다; 아래의 표는 다양한 배포판을 요약한 것입니다.

배포판	파생된 곳	해	소유자	GPL 호환?
0.9.0 ~ 1.2	n/a	1991-1995	CWI	yes
1.3 ~ 1.5.2	1.2	1995-1999	CNRI	yes
1.6	1.5.2	2000	CNRI	no
2.0	1.6	2000	BeOpen.com	no
1.6.1	1.6	2001	CNRI	no
2.1	2.0+1.6.1	2001	PSF	no
2.0.1	2.0+1.6.1	2001	PSF	yes
2.1.1	2.1+2.0.1	2001	PSF	yes
2.1.2	2.1.1	2002	PSF	yes
2.1.3	2.1.2	2002	PSF	yes
2.2 이상	2.1.1	2001-현재	PSF	yes

참고: GPL과 호환된다는 것은 우리가 GPL로 파이썬을 배포한다는 것을 의미하지는 않습니다. 모든 파이썬 라이선스는 GPL과 달리 여러분의 변경을 공개 소스로 만들지 않고 수정된 버전을 배포할 수 있게 합니다. GPL 호환 라이선스는 파이썬과 GPL 하에 발표된 다른 소프트웨어를 결합할 수 있게 합니다; 다른 것들은 그렇지 않습니다.

Guido의 지도하에 이 배포를 가능하게 만든 많은 외부 자원봉사자들에게 감사드립니다.

C.2 파이썬에 액세스하거나 사용하기 위한 이용 약관

파이썬 소프트웨어와 설명서는 *PSF License Agreement*에 따라 라이선스가 부여됩니다.

파이썬 3.8.6부터, 설명서의 예제, 조리법 및 기타 코드는 PSF License Agreement와 *Zero-Clause BSD license*에 따라 이중 라이선스가 부여됩니다.

파이썬에 통합된 일부 소프트웨어에는 다른 라이선스가 적용됩니다. 라이선스는 해당 라이선스에 해당하는 코드와 함께 나열됩니다. 이러한 라이선스의 불완전한 목록은 포함된 소프트웨어에 대한 라이선스 및 승인을 참조하십시오.

C.2.1 PSF LICENSE AGREEMENT FOR PYTHON 3.8.18

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"),
→and
the Individual or Organization ("Licensee") accessing and otherwise using.
→Python
3.8.18 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 3.8.18 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 3.8.18 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 3.8.18 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
3.8.18.
4. PSF is making Python 3.8.18 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 3.8.18 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.8.18
 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT
 ↳OF
 MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.8.18, 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 3.8.18, Licensee agrees
 to be bound by the terms and conditions of this License Agreement.

C.2.2 BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0

BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at 160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization ("Licensee") accessing and otherwise using this software in source or binary form and its associated documentation ("the Software").
2. Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.
3. BeOpen is making the Software available to Licensee on an "AS IS" basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
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.

(다음 페이지에 계속)

(이전 페이지에서 계속)

5. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
6. This License Agreement shall be governed by and interpreted in all respects by the law of the State of California, excluding conflict of law provisions. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between BeOpen and Licensee. This License Agreement does not grant permission to use BeOpen trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party. As an exception, the "BeOpen Python" logos available at <http://www.pythonlabs.com/logos.html> may be used according to the permissions granted on that web page.
7. By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

C.2.3 CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1

1. This LICENSE AGREEMENT is between the Corporation for National Research Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191 ("CNRI"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 1.6.1 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, CNRI hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 1.6.1 alone or in any derivative version, provided, however, that CNRI's License Agreement and CNRI's notice of copyright, i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version prepared by Licensee. Alternately, in lieu of CNRI's License Agreement, Licensee may substitute the following text (omitting the quotes): "Python 1.6.1 is made available subject to the terms and conditions in CNRI's License Agreement. This Agreement together with Python 1.6.1 may be located on the Internet using the following unique, persistent identifier (known as a handle): 1895.22/1013. This Agreement may also be obtained from a proxy server on the Internet using the following URL: <http://hdl.handle.net/1895.22/1013>."
3. In the event Licensee prepares a derivative work that is based on or incorporates Python 1.6.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 1.6.1.
4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis. CNRI MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
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

(다음 페이지에 계속)

(이전 페이지에서 계속)

THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. This License Agreement shall be governed by the federal intellectual property law of the United States, including without limitation the federal copyright law, and, to the extent such U.S. federal law does not apply, by the law of the Commonwealth of Virginia, excluding Virginia's conflict of law provisions. Notwithstanding the foregoing, with regard to derivative works based on Python 1.6.1 that incorporate non-separable material that was previously distributed under the GNU General Public License (GPL), the law of the Commonwealth of Virginia shall govern this License Agreement only as to issues arising under or with respect to Paragraphs 4, 5, and 7 of this License Agreement. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between CNRI and Licensee. This License Agreement does not grant permission to use CNRI trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.
8. By clicking on the "ACCEPT" button where indicated, or by copying, installing or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

C.2.4 CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The Netherlands. All rights reserved.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Stichting Mathematisch Centrum or CWI not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

C.2.5 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON 3.8.18 DOCUMENTATION

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 포함된 소프트웨어에 대한 라이선스 및 승인

이 섹션은 파이썬 배포판에 포함된 제삼자 소프트웨어에 대한 불완전하지만 늘어나고 있는 라이선스와 승인의 목록입니다.

C.3.1 메르센 트위스터

_random 모듈은 <http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html> 에서 내려 받은 코드에 기반한 코드를 포함합니다. 다음은 원래 코드의 주석을 그대로 옮긴 것입니다:

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,

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
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 소켓

socket 모듈은 `getaddrinfo()`와 `getnameinfo()` 함수를 사용합니다. 이들은 WIDE Project, <http://www.wide.ad.jp/>, 에서 온 별도 소스 파일로 코딩되어 있습니다.

Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

C.3.3 비동기 소켓 서비스

asynchat과 asyncore 모듈은 다음과 같은 주의 사항을 포함합니다:

```
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 쿠키 관리

http.cookies 모듈은 다음과 같은 주의 사항을 포함합니다:

```
Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>
```

```
    All Rights Reserved
```

```
Permission to use, copy, modify, and distribute this software
and its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Timothy O'Malley not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.
```

```
Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR
ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3.5 실행 추적

trace 모듈은 다음과 같은 주의 사항을 포함합니다:

```
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 및 UUdecode 함수

uu 모듈은 다음과 같은 주의 사항을 포함합니다:

```
Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
    All Rights Reserved
Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. The C
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
version is still 5 times faster, though.  
- Arguments more compliant with Python standard
```

C.3.7 XML 원격 프로시저 호출

xmlrpc.client 모듈은 다음과 같은 주의 사항을 포함합니다:

```
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 MERCHANT-  
ABILITY AND FITNESS. IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR  
BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY  
DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,  
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS  
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE  
OF THIS SOFTWARE.
```

C.3.8 test_epoll

test_epoll 모듈은 다음과 같은 주의 사항을 포함합니다:

```
Copyright (c) 2001-2006 Twisted Matrix Laboratories.  
  
Permission is hereby granted, free of charge, to any person obtaining  
a copy of this software and associated documentation files (the  
"Software"), to deal in the Software without restriction, including  
without limitation the rights to use, copy, modify, merge, publish,  
distribute, sublicense, and/or sell copies of the Software, and to  
permit persons to whom the Software is furnished to do so, subject to  
the following conditions:  
  
The above copyright notice and this permission notice shall be  
included in all copies or substantial portions of the Software.  
  
THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,  
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.9 Select kqueue

select 모듈은 kqueue 인터페이스에 대해 다음과 같은 주의 사항을 포함합니다:

```
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

파일 Python/pyhash.c 에는 Dan Bernstein의 SipHash24 알고리즘의 Marek Majkowski의 구현이 포함되어 있습니다. 여기에는 다음과 같은 내용이 포함되어 있습니다:

```
<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>
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
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 와 dtoa

C double과 문자열 간의 변환을 위한 C 함수 `dtoa`와 `strtod`를 제공하는 파일 `Python/dtoa.c`는 현재 <http://www.netlib.org/fp/>에서 얻을 수 있는 David M. Gay의 같은 이름의 파일에서 파생되었습니다. 2009년 3월 16일에 받은 원본 파일에는 다음과 같은 저작권 및 라이선스 공지가 포함되어 있습니다:

```
/* *****
 *
 * 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

모듈 `hashlib`, `posix`, `ssl`, `crypt`는 운영 체제가 사용할 수 있게 하면 추가의 성능을 위해 OpenSSL 라이브러리를 사용합니다. 또한, 윈도우와 맥 OS X 파이썬 설치 프로그램은 OpenSSL 라이브러리 사본을 포함할 수 있으므로, 여기에 OpenSSL 라이선스 사본을 포함합니다:

```
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.
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```

*
* 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. 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)

(다음 페이지에 계속)

(이전 페이지에서 계속)

```

* 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,
* 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 rouines 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 확장은 빌드를 `--with-system-expat` 로 구성하지 않는 한, 포함된 expat 소스 사본을 사용하여 빌드됩니다:

```
Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd
and Clark Cooper

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.14 libffi

_ctypes 확장은 빌드를 `--with-system-libffi` 로 구성하지 않는 한, 포함된 libffi 소스 사본을 사용하여 빌드됩니다:

```
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 확장은 시스템에서 발견된 zlib 버전이 너무 오래되어서 빌드에 사용될 수 없으면, 포함된 zlib 소스 사본을 사용하여 빌드됩니다:

```
Copyright (C) 1995-2011 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied
warranty. In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.

3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly          Mark Adler
jloup@gzip.org            madler@alumni.caltech.edu
```

C.3.16 cfuhash

tracemalloc에 의해 사용되는 해시 테이블의 구현은 cfuhash 프로젝트를 기반으로 합니다:

```
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
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
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` 모듈은 빌드를 `--with-system-libmpdec` 로 구성하지 않는 한, 포함된 `libmpdec` 소스 사본을 사용하여 빌드됩니다:

```
Copyright (c) 2008-2016 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 테스트 스위트

`test` 패키지의 C14N 2.0 테스트 스위트(`Lib/test/xmltestdata/c14n-20/`)는 W3C 웹 사이트 <https://www.w3.org/TR/xml-c14n2-testcases/> 에서 가져왔으며 3-절 BSD 라이선스 하에 배포됩니다:

Copyright (c) 2013 W3C(R) (MIT, ERCIM, Keio, Beihang), All Rights Reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of works must retain the original copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the original copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

- Neither the name of the W3C nor the names of its contributors may be used to endorse or promote products derived from this work without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

APPENDIX D

저작권

파이썬과 이 설명서는:

Copyright © 2001-2023 Python Software Foundation. All rights reserved.

Copyright © 2000 BeOpen.com. All rights reserved.

Copyright © 1995-2000 Corporation for National Research Initiatives. All rights reserved.

Copyright © 1991-1995 Stichting Mathematisch Centrum. All rights reserved.

전체 라이선스 및 사용 권한 정보는 [역사](#)와 [라이선스](#) 에서 제공합니다.

Non-alphabetical

..., 229
 2to3, 229
 >>>, 229
 __all__ (package variable), 40
 __dict__ (module attribute), 121
 __doc__ (module attribute), 121
 __file__ (module attribute), 121, 122
 __future__, 233
 __import__
 내장 함수, 40
 __loader__ (module attribute), 121
 __main__
 모듈, 11, 142, 152
 __name__ (module attribute), 121, 122
 __package__ (module attribute), 121
 __slots__, 239
 _frozen (C 데이터 형식), 43
 _inittab (C 데이터 형식), 43
 _Py_c_diff (C 함수), 85
 _Py_c_neg (C 함수), 85
 _Py_c_pow (C 함수), 85
 _Py_c_prod (C 함수), 85
 _Py_c_quot (C 함수), 85
 _Py_c_sum (C 함수), 85
 _Py_InitializeMain (C 함수), 173
 _Py_NoneStruct (C 변수), 186
 _Py_TPFLAGS_HAVE_VECTORCALL (내장 변수), 204
 _PyBytes_Resize (C 함수), 88
 _PyCFunctionFast (C 데이터 형식), 187
 _PyCFunctionFastWithKeywords (C 데이터 형식), 187
 _PyImport_Fini (C 함수), 43
 _PyImport_Init (C 함수), 42
 _PyObject_FastCallDict (C 함수), 61
 _PyObject_New (C 함수), 185
 _PyObject_NewVar (C 함수), 185
 _PyObject_Vectorcall (C 함수), 61

_PyTuple_Resize (C 함수), 109

_thread

 모듈, 148

객체

bytearray, 88
 bytes, 86
 Capsule, 131
 complex number, 85
 dictionary, 112
 file, 120
 floating point, 84
 frozenset, 115
 function, 117
 instancemethod, 118
 integer, 80
 list, 111
 long integer, 80
 mapping, 112
 memoryview, 130
 method, 118
 module, 121
 None, 80
 numeric, 80
 sequence, 86
 set, 115
 tuple, 108
 type, 6, 77

A

abort(), 40
 abs
 내장 함수, 64
 abstract base class (추상 베이스 클래스), 229
 allocfunc (C 데이터 형식), 220
 annotation (어노테이션), 229
 argument (인자), 229
 argv (in module sys), 145
 ascii
 내장 함수, 59

asynchronous context manager (비동기 컨텍스트 관리자), **230**
 asynchronous generator (비동기 제너레이터), **230**
 asynchronous generator iterator (비동기 제너레이터 이터레이터), **230**
 asynchronous iterable (비동기 이터러블), **230**
 asynchronous iterator (비동기 이터레이터), **230**
 attribute (어트리뷰트), **230**
 awaitable (어웨이터블), **230**

B

BDFL, **230**
 binary file (바이너리 파일), **230**
 binaryfunc (C 데이터 형식), **221**
 buffer interface
 (see buffer protocol), **69**
 buffer object
 (see buffer protocol), **69**
 buffer protocol, **69**
 builtins
 모듈, **11**, **142**, **152**
 bytearray
 객체, **88**
 bytecode (바이트 코드), **231**
 bytes
 객체, **86**
 내장 함수, **59**
 bytes-like object (바이트열류 객체), **230**

C

callback (콜백), **231**
 calloc(), **175**
 Capsule
 객체, **131**
 C-contiguous, **72**, **231**
 class (클래스), **231**
 class variable (클래스 변수), **231**
 classmethod
 내장 함수, **188**
 cleanup functions, **40**
 close() (in module os), **153**
 CO_FUTURE_DIVISION (C 변수), **19**
 code object, **119**
 coercion (코어션), **231**
 compile
 내장 함수, **41**
 complex number
 객체, **85**
 complex number (복소수), **231**
 context manager (컨텍스트 관리자), **231**
 context variable (컨텍스트 변수), **231**
 contiguous, **72**

contiguous (연속), **231**
 copyright (in module sys), **145**
 coroutine (코루틴), **231**
 coroutine function (코루틴 함수), **231**
 CPython, **231**
 create_module (C 함수), **124**

D

decorator (데코레이터), **231**
 descrgetfunc (C 데이터 형식), **220**
 descriptor (디스크립터), **232**
 descrsetfunc (C 데이터 형식), **220**
 destructor (C 데이터 형식), **220**
 dictionary
 객체, **112**
 dictionary (딕셔너리), **232**
 dictionary comprehension, **232**
 dictionary view (딕셔너리 뷰), **232**
 divmod
 내장 함수, **63**
 docstring (독스트링), **232**
 duck-typing (덕 타이핑), **232**

E

EAFP, **232**
 EOFError (built-in exception), **120**
 exc_info() (in module sys), **10**
 exec_module (C 함수), **125**
 exec_prefix, **4**
 executable (in module sys), **144**
 exit(), **40**
 expression (표현식), **232**
 extension module (확장 모듈), **232**

F

f-string (f-문자열), **232**
 file
 객체, **120**
 file object (파일 객체), **233**
 file-like object (파일류 객체), **233**
 finder (파인더), **233**
 float
 내장 함수, **65**
 floating point
 객체, **84**
 floor division (정수 나눗셈), **233**
 Fortran contiguous, **72**, **231**
 free(), **175**
 freefunc (C 데이터 형식), **220**
 freeze utility, **43**
 frozenset
 객체, **115**
 function
 객체, **117**

function (함수), **233**
 function annotation (함수 어노테이션), **233**

G

garbage collection (가비지 수거), **233**
 generator, **233**
 generator (제너레이터), **233**
 generator expression, **233**
 generator expression (제너레이터 표현식), **234**
 generator iterator (제너레이터 이터레이터), **233**
 generic function (제네릭 함수), **234**
 getattrfunc (C 데이터 형식), **220**
 getattrofunc (C 데이터 형식), **220**
 getbufferproc (C 데이터 형식), **221**
 getiterfunc (C 데이터 형식), **221**
 GIL, **234**
 global interpreter lock, **146**
 global interpreter lock (전역 인터프리터 락), **234**

H

hash
 내장 함수, **62, 201**
 hash-based pyc (해시 기반 pyc), **234**
 hashable (해시 가능), **234**
 hashfunc (C 데이터 형식), **220**

I

IDLE, **234**
 immutable (불변), **234**
 import path (임포트 경로), **234**
 importer (임porter), **234**
 importing (임포트), **234**
 incr_item(), **10, 11**
 initproc (C 데이터 형식), **220**
 inquiry (C 데이터 형식), **225**
 instancemethod
 객체, **118**
 int
 내장 함수, **65**
 integer
 객체, **80**
 interactive (대화형), **234**
 interpreted (인터프리티드), **235**
 interpreter lock, **146**
 interpreter shutdown (인터프리터 종료), **235**
 iterable (이터러블), **235**
 iterator (이터레이터), **235**
 iternextfunc (C 데이터 형식), **221**

K

key function (키 함수), **235**

KeyboardInterrupt (*built-in exception*), **29**
 keyword argument (키워드 인자), **235**

L

lambda (람다), **235**
 LBYL, **235**
 len
 내장 함수, **62, 66, 68, 111, 114, 116**
 lenfunc (C 데이터 형식), **221**
 list
 객체, **111**
 list (리스트), **236**
 list comprehension (리스트 컴프리헨션), **236**
 loader (로더), **236**
 lock, interpreter, **146**
 long integer
 객체, **80**
 LONG_MAX, **81**

M

magic
 method, **236**
 magic method (매직 메서드), **236**
 main(), **143, 145**
 malloc(), **175**
 mapping
 객체, **112**
 mapping (매핑), **236**
 memoryview
 객체, **130**
 meta path finder (메타 경로 파인더), **236**
 metaclass (메타 클래스), **236**
 METH_CLASS (내장 변수), **188**
 METH_COEXIST (내장 변수), **188**
 METH_FASTCALL (내장 변수), **188**
 METH_NOARGS (내장 변수), **188**
 METH_O (내장 변수), **188**
 METH_STATIC (내장 변수), **188**
 METH_VARARGS (내장 변수), **187**
 method
 magic, **236**
 special, **240**
 객체, **118**
 method (메서드), **236**
 method resolution order (메서드 결정 순서), **236**
 MethodType (*in module types*), **117, 118**
 module
 search path, **11, 142, 144**
 객체, **121**
 module (모듈), **236**
 module spec (모듈 스펙), **236**
 modules (*in module sys*), **40, 142**
 ModuleType (*in module types*), **121**

MRO, 236

mutable (가변), 236

N

named tuple (네임드 튜플), 236

namespace (이름 공간), 237

namespace package (이름 공간 패키지), 237

nested scope (중첩된 스코프), 237

new-style class (뉴스타일 클래스), 237

newfunc (C 데이터 형식), 220

None

객체, 80

numeric

객체, 80

O

object

code, 119

object (객체), 237

objobjargproc (C 데이터 형식), 221

objobjproc (C 데이터 형식), 221

OverflowError (built-in exception), 8183

P

package (패키지), 237

package variable

__all__, 40

parameter (매개변수), 237

PATH, 11

path

module search, 11, 142, 144

path (in module sys), 11, 142, 144

path based finder (경로 기반 파인더), 238

path entry (경로 엔트리), 238

path entry finder (경로 엔트리 파인더), 238

path entry hook (경로 엔트리 훅), 238

path-like object (경로류 객체), 238

PEP, 238

platform (in module sys), 144

portion (포션), 238

positional argument (위치 인자), 238

pow

내장 함수, 63, 65

prefix, 4

provisional API (잠정 API), 238

provisional package (잠정 패키지), 239

Py_ABS (C 매크로), 4

Py_AddPendingCall (C 함수), 153

Py_AddPendingCall(), 153

Py_AtExit (C 함수), 40

Py_BEGIN_ALLOW_THREADS, 146

Py_BEGIN_ALLOW_THREADS (C 매크로), 150

Py_BLOCK_THREADS (C 매크로), 150

Py_buffer (C 데이터 형식), 70

Py_buffer.buf (C 멤버 변수), 70

Py_buffer.format (C 멤버 변수), 71

Py_buffer.internal (C 멤버 변수), 71

Py_buffer.itemsize (C 멤버 변수), 70

Py_buffer.len (C 멤버 변수), 70

Py_buffer.ndim (C 멤버 변수), 71

Py_buffer.obj (C 멤버 변수), 70

Py_buffer.readonly (C 멤버 변수), 70

Py_buffer.shape (C 멤버 변수), 71

Py_buffer.strides (C 멤버 변수), 71

Py_buffer.suboffsets (C 멤버 변수), 71

Py_BuildValue (C 함수), 51

Py_BytesMain (C 함수), 15

Py_BytesWarningFlag (C 변수), 140

Py_CHARMASK (C 매크로), 5

Py_CLEAR (C 함수), 21

Py_CompileString (C 함수), 18

Py_CompileString(), 19

Py_CompileStringExFlags (C 함수), 18

Py_CompileStringFlags (C 함수), 18

Py_CompileStringObject (C 함수), 18

Py_complex (C 데이터 형식), 85

Py_DebugFlag (C 변수), 140

Py_DecodeLocale (C 함수), 36

Py_DECREF (C 함수), 21

Py_DECREF(), 6

Py_DEPRECATED (C 매크로), 5

Py_DontWriteBytecodeFlag (C 변수), 140

Py_Ellipsis (C 변수), 130

Py_EncodeLocale (C 함수), 37

Py_END_ALLOW_THREADS, 146

Py_END_ALLOW_THREADS (C 매크로), 150

Py_EndInterpreter (C 함수), 153

Py_EnterRecursiveCall (C 함수), 32

Py_eval_input (C 변수), 19

Py_Exit (C 함수), 40

Py_ExitStatusException (C 함수), 161

Py_False (C 변수), 83

Py_FatalError (C 함수), 40

Py_FatalError(), 145

Py_FdIsInteractive (C 함수), 35

Py_file_input (C 변수), 19

Py_Finalize (C 함수), 143

Py_FinalizeEx (C 함수), 142

Py_FinalizeEx(), 40, 142, 152, 153

Py_FrozenFlag (C 변수), 140

Py_GetBuildInfo (C 함수), 145

Py_GetCompiler (C 함수), 145

Py_GetCopyright (C 함수), 145

Py_GETENV (C 매크로), 5

Py_GetExecPrefix (C 함수), 143

Py_GetExecPrefix(), 11

Py_GetPath (C 함수), 144

Py_GetPath(), 11, 143, 144

- Py_GetPlatform (C 함수), 144
- Py_GetPrefix (C 함수), 143
- Py_GetPrefix(), 11
- Py_GetProgramFullPath (C 함수), 144
- Py_GetProgramFullPath(), 11
- Py_GetProgramName (C 함수), 143
- Py_GetPythonHome (C 함수), 146
- Py_GetVersion (C 함수), 144
- Py_HashRandomizationFlag (C 변수), 140
- Py_IgnoreEnvironmentFlag (C 변수), 141
- Py_INCREF (C 함수), 21
- Py_INCREF(), 6
- Py_Initialize (C 함수), 142
- Py_Initialize(), 11, 143, 152
- Py_InitializeEx (C 함수), 142
- Py_InitializeFromConfig (C 함수), 168
- Py_InspectFlag (C 변수), 141
- Py_InteractiveFlag (C 변수), 141
- Py_IsInitialized (C 함수), 142
- Py_IsInitialized(), 12
- Py_IsolatedFlag (C 변수), 141
- Py_LeaveRecursiveCall (C 함수), 32
- Py_LegacyWindowsFSEncodingFlag (C 변수), 141
- Py_LegacyWindowsStdioFlag (C 변수), 141
- Py_Main (C 함수), 15
- Py_MAX (C 매크로), 5
- Py_MEMBER_SIZE (C 매크로), 5
- Py_MIN (C 매크로), 5
- Py_mod_create (C 매크로), 124
- Py_mod_exec (C 매크로), 125
- Py_NewInterpreter (C 함수), 152
- Py_None (C 변수), 80
- Py_NoSiteFlag (C 변수), 141
- Py_NotImplemented (C 변수), 57
- Py_NoUserSiteDirectory (C 변수), 141
- Py_OptimizeFlag (C 변수), 141
- Py_PreInitialize (C 함수), 163
- Py_PreInitializeFromArgs (C 함수), 163
- Py_PreInitializeFromBytesArgs (C 함수), 163
- Py_PRINT_RAW, 121
- Py_QuietFlag (C 변수), 141
- Py_REFCNT (C 매크로), 186
- Py_ReprEnter (C 함수), 32
- Py_ReprLeave (C 함수), 32
- Py_RETURN_FALSE (C 매크로), 84
- Py_RETURN_NONE (C 매크로), 80
- Py_RETURN_NOTIMPLEMENTED (C 매크로), 57
- Py_RETURN_RICHCOMPARE (C 매크로), 207
- Py_RETURN_TRUE (C 매크로), 84
- Py_RunMain (C 함수), 172
- Py_SetPath (C 함수), 144
- Py_SetPath(), 144
- Py_SetProgramName (C 함수), 143
- Py_SetProgramName(), 11, 142, 144
- Py_SetPythonHome (C 함수), 146
- Py_SetStandardStreamEncoding (C 함수), 143
- Py_single_input (C 변수), 19
- Py_SIZE (C 매크로), 186
- PY_SSIZE_T_MAX, 82
- Py_STRINGIFY (C 매크로), 5
- Py_TPFLAGS_BASE_EXC_SUBCLASS (내장 변수), 204
- Py_TPFLAGS_BASETYPE (내장 변수), 203
- Py_TPFLAGS_BYTES_SUBCLASS (내장 변수), 204
- Py_TPFLAGS_DEFAULT (내장 변수), 203
- Py_TPFLAGS_DICT_SUBCLASS (내장 변수), 204
- Py_TPFLAGS_HAVE_FINALIZE (내장 변수), 204
- Py_TPFLAGS_HAVE_GC (내장 변수), 203
- Py_TPFLAGS_HEAPTYPE (내장 변수), 203
- Py_TPFLAGS_LIST_SUBCLASS (내장 변수), 204
- Py_TPFLAGS_LONG_SUBCLASS (내장 변수), 204
- Py_TPFLAGS_METHOD_DESCRIPTOR (내장 변수), 204
- Py_TPFLAGS_READY (내장 변수), 203
- Py_TPFLAGS_READYING (내장 변수), 203
- Py_TPFLAGS_TUPLE_SUBCLASS (내장 변수), 204
- Py_TPFLAGS_TYPE_SUBCLASS (내장 변수), 204
- Py_TPFLAGS_UNICODE_SUBCLASS (내장 변수), 204
- Py_tracefunc (C 데이터 형식), 154
- Py_True (C 변수), 84
- Py_tss_NEEDS_INIT (C 매크로), 156
- Py_tss_t (C 데이터 형식), 156
- Py_TYPE (C 매크로), 186
- Py_UCS1 (C 데이터 형식), 89
- Py_UCS2 (C 데이터 형식), 89
- Py_UCS4 (C 데이터 형식), 89
- Py_UNBLOCK_THREADS (C 매크로), 150
- Py_UnbufferedStdioFlag (C 변수), 141
- Py_UNICODE (C 데이터 형식), 89
- Py_UNICODE_IS_HIGH_SURROGATE (C 매크로), 93
- Py_UNICODE_IS_LOW_SURROGATE (C 매크로), 93
- Py_UNICODE_IS_SURROGATE (C 매크로), 93
- Py_UNICODE_ISALNUM (C 함수), 92
- Py_UNICODE_ISALPHA (C 함수), 92
- Py_UNICODE_ISDECIMAL (C 함수), 92
- Py_UNICODE_ISDIGIT (C 함수), 92
- Py_UNICODE_ISLINEBREAK (C 함수), 92
- Py_UNICODE_ISLOWER (C 함수), 92
- Py_UNICODE_ISNUMERIC (C 함수), 92
- Py_UNICODE_ISPRINTABLE (C 함수), 92
- Py_UNICODE_ISSPACE (C 함수), 92
- Py_UNICODE_ISTITLE (C 함수), 92
- Py_UNICODE_ISUPPER (C 함수), 92
- Py_UNICODE_JOIN_SURROGATES (C 매크로), 93
- Py_UNICODE_TODECIMAL (C 함수), 93
- Py_UNICODE_TODIGIT (C 함수), 93

- Py_UNICODE_TOLOWER (C 함수), 92
- Py_UNICODE_TONUMERIC (C 함수), 93
- Py_UNICODE_TOTITLE (C 함수), 93
- Py_UNICODE_TOUPPER (C 함수), 92
- Py_UNREACHABLE (C 매크로), 4
- Py_UNUSED (C 매크로), 5
- Py_VaBuildValue (C 함수), 52
- Py_VECTORCALL_ARGUMENTS_OFFSET (C 매크로), 61
- Py_VerboseFlag (C 변수), 142
- Py_VISIT (C 함수), 225
- Py_XDECREF (C 함수), 21
- Py_XDECREF (), 11
- Py_XINCRREF (C 함수), 21
- PyAnySet_Check (C 함수), 115
- PyAnySet_CheckExact (C 함수), 115
- PyArg_Parse (C 함수), 50
- PyArg_ParseTuple (C 함수), 49
- PyArg_ParseTupleAndKeywords (C 함수), 49
- PyArg_UnpackTuple (C 함수), 50
- PyArg_ValidateKeywordArguments (C 함수), 50
- PyArg_VaParse (C 함수), 49
- PyArg_VaParseTupleAndKeywords (C 함수), 49
- PyASCIIObject (C 데이터 형식), 90
- PyAsyncMethods (C 데이터 형식), 219
- PyAsyncMethods.am_aiter (C 멤버 변수), 219
- PyAsyncMethods.am_anext (C 멤버 변수), 219
- PyAsyncMethods.am_await (C 멤버 변수), 219
- PyBool_Check (C 함수), 83
- PyBool_FromLong (C 함수), 84
- PyBUF_ANY_CONTIGUOUS (C 매크로), 73
- PyBUF_C_CONTIGUOUS (C 매크로), 73
- PyBUF_CONTIG (C 매크로), 73
- PyBUF_CONTIG_RO (C 매크로), 73
- PyBUF_F_CONTIGUOUS (C 매크로), 73
- PyBUF_FORMAT (C 매크로), 72
- PyBUF_FULL (C 매크로), 73
- PyBUF_FULL_RO (C 매크로), 73
- PyBUF_INDIRECT (C 매크로), 72
- PyBUF_ND (C 매크로), 72
- PyBUF_RECORDS (C 매크로), 73
- PyBUF_RECORDS_RO (C 매크로), 73
- PyBUF_SIMPLE (C 매크로), 72
- PyBUF_STRIDED (C 매크로), 73
- PyBUF_STRIDED_RO (C 매크로), 73
- PyBUF_STRIDES (C 매크로), 72
- PyBUF_WRITABLE (C 매크로), 72
- PyBuffer_FillContiguousStrides (C 함수), 75
- PyBuffer_FillInfo (C 함수), 76
- PyBuffer_FromContiguous (C 함수), 75
- PyBuffer_GetPointer (C 함수), 75
- PyBuffer_IsContiguous (C 함수), 75
- PyBuffer_Release (C 함수), 75
- PyBuffer_SizeFromFormat (C 함수), 75
- PyBuffer_ToContiguous (C 함수), 75
- PyBufferProcs, 69
- PyBufferProcs (C 데이터 형식), 218
- PyBufferProcs.bf_getbuffer (C 멤버 변수), 218
- PyBufferProcs.bf_releasebuffer (C 멤버 변수), 218
- PyByteArray_AS_STRING (C 함수), 89
- PyByteArray_AsString (C 함수), 88
- PyByteArray_Check (C 함수), 88
- PyByteArray_CheckExact (C 함수), 88
- PyByteArray_Concat (C 함수), 88
- PyByteArray_FromObject (C 함수), 88
- PyByteArray_FromStringAndSize (C 함수), 88
- PyByteArray_GET_SIZE (C 함수), 89
- PyByteArray_Resize (C 함수), 89
- PyByteArray_Size (C 함수), 88
- PyByteArray_Type (C 변수), 88
- PyByteArrayObject (C 데이터 형식), 88
- PyBytes_AS_STRING (C 함수), 87
- PyBytes_AsString (C 함수), 87
- PyBytes_AsStringAndSize (C 함수), 87
- PyBytes_Check (C 함수), 86
- PyBytes_CheckExact (C 함수), 86
- PyBytes_Concat (C 함수), 88
- PyBytes_ConcatAndDel (C 함수), 88
- PyBytes_FromFormat (C 함수), 86
- PyBytes_FromFormatV (C 함수), 87
- PyBytes_FromObject (C 함수), 87
- PyBytes_FromString (C 함수), 86
- PyBytes_FromStringAndSize (C 함수), 86
- PyBytes_GET_SIZE (C 함수), 87
- PyBytes_Size (C 함수), 87
- PyBytes_Type (C 변수), 86
- PyBytesObject (C 데이터 형식), 86
- PyCallable_Check (C 함수), 60
- PyCallIter_Check (C 함수), 127
- PyCallIter_New (C 함수), 127
- PyCallIter_Type (C 변수), 127
- PyCapsule (C 데이터 형식), 131
- PyCapsule_CheckExact (C 함수), 131
- PyCapsule_Destructor (C 데이터 형식), 131
- PyCapsule_GetContext (C 함수), 132
- PyCapsule_GetDestructor (C 함수), 132
- PyCapsule_GetName (C 함수), 132
- PyCapsule_GetPointer (C 함수), 132
- PyCapsule_Import (C 함수), 132
- PyCapsule_IsValid (C 함수), 132
- PyCapsule_New (C 함수), 131
- PyCapsule_SetContext (C 함수), 132
- PyCapsule_SetDestructor (C 함수), 132
- PyCapsule_SetName (C 함수), 132

- PyCapsule_SetPointer (C 함수), 133
- PyCell_Check (C 함수), 119
- PyCell_GET (C 함수), 119
- PyCell_Get (C 함수), 119
- PyCell_New (C 함수), 119
- PyCell_SET (C 함수), 119
- PyCell_Set (C 함수), 119
- PyCell_Type (C 변수), 119
- PyCellobject (C 데이터 형식), 119
- PyCFunction (C 데이터 형식), 187
- PyCFunctionWithKeywords (C 데이터 형식), 187
- PyCode_Check (C 함수), 119
- PyCode_GetNumFree (C 함수), 119
- PyCode_New (C 함수), 120
- PyCode_NewEmpty (C 함수), 120
- PyCode_NewWithPosOnlyArgs (C 함수), 120
- PyCode_Type (C 변수), 119
- PyCodec_BackslashReplaceErrors (C 함수), 56
- PyCodec_Decompile (C 함수), 54
- PyCodec_Decoder (C 함수), 55
- PyCodec_Encode (C 함수), 54
- PyCodec_Encoder (C 함수), 55
- PyCodec_IgnoreErrors (C 함수), 55
- PyCodec_IncrementalDecoder (C 함수), 55
- PyCodec_IncrementalEncoder (C 함수), 55
- PyCodec_KnownEncoding (C 함수), 54
- PyCodec_LookupError (C 함수), 55
- PyCodec_NameReplaceErrors (C 함수), 56
- PyCodec_Register (C 함수), 54
- PyCodec_RegisterError (C 함수), 55
- PyCodec_ReplaceErrors (C 함수), 55
- PyCodec_StreamReader (C 함수), 55
- PyCodec_StreamWriter (C 함수), 55
- PyCodec_StrictErrors (C 함수), 55
- PyCodec_XMLCharRefReplaceErrors (C 함수), 55
- PyCodeObject (C 데이터 형식), 119
- PyCompactUnicodeObject (C 데이터 형식), 90
- PyCompilerFlags (C 데이터 형식), 19
- PyCompilerFlags.cf_feature_version (C 멤버 변수), 19
- PyCompilerFlags.cf_flags (C 멤버 변수), 19
- PyComplex_AsCComplex (C 함수), 86
- PyComplex_Check (C 함수), 85
- PyComplex_CheckExact (C 함수), 85
- PyComplex_FromCComplex (C 함수), 86
- PyComplex_FromDoubles (C 함수), 86
- PyComplex_ImagAsDouble (C 함수), 86
- PyComplex_RealAsDouble (C 함수), 86
- PyComplex_Type (C 변수), 85
- PyComplexObject (C 데이터 형식), 85
- PyConfig (C 데이터 형식), 164
- PyConfig_Clear (C 함수), 165
- PyConfig_InitIsolatedConfig (C 함수), 164
- PyConfig_InitPythonConfig (C 함수), 164
- PyConfig_Read (C 함수), 164
- PyConfig_SetArgv (C 함수), 164
- PyConfig_SetBytesArgv (C 함수), 164
- PyConfig_SetBytesString (C 함수), 164
- PyConfig_SetString (C 함수), 164
- PyConfig_SetWideStringList (C 함수), 164
- PyConfig.argv (C 멤버 변수), 165
- PyConfig.base_exec_prefix (C 멤버 변수), 165
- PyConfig.base_executable (C 멤버 변수), 165
- PyConfig.base_prefix (C 멤버 변수), 165
- PyConfig.buffered_stdio (C 멤버 변수), 165
- PyConfig.bytes_warning (C 멤버 변수), 165
- PyConfig.check_hash_pycs_mode (C 멤버 변수), 165
- PyConfig.configure_c_stdio (C 멤버 변수), 165
- PyConfig.dev_mode (C 멤버 변수), 165
- PyConfig.dump_refs (C 멤버 변수), 166
- PyConfig.exec_prefix (C 멤버 변수), 166
- PyConfig.executable (C 멤버 변수), 166
- PyConfig.faulthandler (C 멤버 변수), 166
- PyConfig.filesystem_encoding (C 멤버 변수), 166
- PyConfig.filesystem_errors (C 멤버 변수), 166
- PyConfig.hash_seed (C 멤버 변수), 166
- PyConfig.home (C 멤버 변수), 166
- PyConfig.import_time (C 멤버 변수), 166
- PyConfig.inspect (C 멤버 변수), 166
- PyConfig.install_signal_handlers (C 멤버 변수), 166
- PyConfig.interactive (C 멤버 변수), 166
- PyConfig.isolated (C 멤버 변수), 166
- PyConfig.legacy_windows_stdio (C 멤버 변수), 166
- PyConfig.malloc_stats (C 멤버 변수), 166
- PyConfig.module_search_paths (C 멤버 변수), 167
- PyConfig.module_search_paths_set (C 멤버 변수), 167
- PyConfig.optimization_level (C 멤버 변수), 167
- PyConfig.parse_argv (C 멤버 변수), 167
- PyConfig.parser_debug (C 멤버 변수), 167
- PyConfig.pathconfig_warnings (C 멤버 변수), 167
- PyConfig.prefix (C 멤버 변수), 167
- PyConfig.program_name (C 멤버 변수), 167
- PyConfig.pycache_prefix (C 멤버 변수), 167
- PyConfig.pythonpath_env (C 멤버 변수), 167
- PyConfig.quiet (C 멤버 변수), 167
- PyConfig.run_command (C 멤버 변수), 167

- PyConfig.run_filename (C 멤버 변수), 167
- PyConfig.run_module (C 멤버 변수), 167
- PyConfig.show_alloc_count (C 멤버 변수), 167
- PyConfig.show_ref_count (C 멤버 변수), 167
- PyConfig.site_import (C 멤버 변수), 168
- PyConfig.skip_source_first_line (C 멤버 변수), 168
- PyConfig.stdio_encoding (C 멤버 변수), 168
- PyConfig.stdio_errors (C 멤버 변수), 168
- PyConfig.tracemalloc (C 멤버 변수), 168
- PyConfig.use_environment (C 멤버 변수), 168
- PyConfig.use_hash_seed (C 멤버 변수), 166
- PyConfig.user_site_directory (C 멤버 변수), 168
- PyConfig.verbose (C 멤버 변수), 168
- PyConfig.warnoptions (C 멤버 변수), 168
- PyConfig.write_bytecode (C 멤버 변수), 168
- PyConfig.xoptions (C 멤버 변수), 168
- PyContext (C 데이터 형식), 134
- PyContext_CheckExact (C 함수), 134
- PyContext_ClearFreeList (C 함수), 135
- PyContext_Copy (C 함수), 134
- PyContext_CopyCurrent (C 함수), 134
- PyContext_Enter (C 함수), 135
- PyContext_Exit (C 함수), 135
- PyContext_New (C 함수), 134
- PyContext_Type (C 변수), 134
- PyContextToken (C 데이터 형식), 134
- PyContextToken_CheckExact (C 함수), 134
- PyContextToken_Type (C 변수), 134
- PyContextVar (C 데이터 형식), 134
- PyContextVar_CheckExact (C 함수), 134
- PyContextVar_Get (C 함수), 135
- PyContextVar_New (C 함수), 135
- PyContextVar_Reset (C 함수), 135
- PyContextVar_Set (C 함수), 135
- PyContextVar_Type (C 변수), 134
- PyCoro_CheckExact (C 함수), 133
- PyCoro_New (C 함수), 133
- PyCoro_Type (C 변수), 133
- PyCoroObject (C 데이터 형식), 133
- PyDate_Check (C 함수), 135
- PyDate_CheckExact (C 함수), 136
- PyDate_FromDate (C 함수), 136
- PyDate_FromTimestamp (C 함수), 138
- PyDateTime_Check (C 함수), 136
- PyDateTime_CheckExact (C 함수), 136
- PyDateTime_DATE_GET_FOLD (C 함수), 137
- PyDateTime_DATE_GET_HOUR (C 함수), 137
- PyDateTime_DATE_GET_MICROSECOND (C 함수), 137
- PyDateTime_DATE_GET_MINUTE (C 함수), 137
- PyDateTime_DATE_GET_SECOND (C 함수), 137
- PyDateTime_DELTA_GET_DAYS (C 함수), 138
- PyDateTime_DELTA_GET_MICROSECONDS (C 함수), 138
- PyDateTime_DELTA_GET_SECONDS (C 함수), 138
- PyDateTime_FromDateAndTime (C 함수), 136
- PyDateTime_FromDateAndTimeAndFold (C 함수), 136
- PyDateTime_FromTimestamp (C 함수), 138
- PyDateTime_GET_DAY (C 함수), 137
- PyDateTime_GET_MONTH (C 함수), 137
- PyDateTime_GET_YEAR (C 함수), 137
- PyDateTime_TIME_GET_FOLD (C 함수), 137
- PyDateTime_TIME_GET_HOUR (C 함수), 137
- PyDateTime_TIME_GET_MICROSECOND (C 함수), 137
- PyDateTime_TIME_GET_MINUTE (C 함수), 137
- PyDateTime_TIME_GET_SECOND (C 함수), 137
- PyDateTime_TimeZone_UTC (C 변수), 135
- PyDelta_Check (C 함수), 136
- PyDelta_CheckExact (C 함수), 136
- PyDelta_FromDSU (C 함수), 136
- PyDescr_IsData (C 함수), 128
- PyDescr_NewClassMethod (C 함수), 128
- PyDescr_NewGetSet (C 함수), 128
- PyDescr_NewMember (C 함수), 128
- PyDescr_NewMethod (C 함수), 128
- PyDescr_NewWrapper (C 함수), 128
- PyDict_Check (C 함수), 112
- PyDict_CheckExact (C 함수), 112
- PyDict_Clear (C 함수), 113
- PyDict_ClearFreeList (C 함수), 115
- PyDict_Contains (C 함수), 113
- PyDict_Copy (C 함수), 113
- PyDict_DelItem (C 함수), 113
- PyDict_DelItemString (C 함수), 113
- PyDict_GetItem (C 함수), 113
- PyDict_GetItemString (C 함수), 113
- PyDict_GetItemWithError (C 함수), 113
- PyDict_Items (C 함수), 114
- PyDict_Keys (C 함수), 114
- PyDict_Merge (C 함수), 114
- PyDict_MergeFromSeq2 (C 함수), 115
- PyDict_New (C 함수), 113
- PyDict_Next (C 함수), 114
- PyDict_SetDefault (C 함수), 113
- PyDict_SetItem (C 함수), 113
- PyDict_SetItemString (C 함수), 113
- PyDict_Size (C 함수), 114
- PyDict_Type (C 변수), 112
- PyDict_Update (C 함수), 114
- PyDict_Values (C 함수), 114
- PyDictObject (C 데이터 형식), 112
- PyDictProxy_New (C 함수), 113
- PyDoc_STR (C 매크로), 5
- PyDoc_STRVAR (C 매크로), 5

- PyErr_BadArgument (C 함수), 24
- PyErr_BadInternalCall (C 함수), 26
- PyErr_CheckSignals (C 함수), 29
- PyErr_Clear (C 함수), 23
- PyErr_Clear(), 9, 11
- PyErr_ExceptionMatches (C 함수), 27
- PyErr_ExceptionMatches(), 11
- PyErr_Fetch (C 함수), 27
- PyErr_Format (C 함수), 24
- PyErr_FormatV (C 함수), 24
- PyErr_GetExcInfo (C 함수), 28
- PyErr_GivenExceptionMatches (C 함수), 27
- PyErr_NewException (C 함수), 29
- PyErr_NewExceptionWithDoc (C 함수), 29
- PyErr_NoMemory (C 함수), 24
- PyErr_NormalizeException (C 함수), 28
- PyErr_Occurred (C 함수), 27
- PyErr_Occurred(), 9
- PyErr_Print (C 함수), 24
- PyErr_PrintEx (C 함수), 23
- PyErr_ResourceWarning (C 함수), 27
- PyErr_Restore (C 함수), 28
- PyErr_SetExcFromWindowsErr (C 함수), 25
- PyErr_SetExcFromWindowsErrWithFilename (C 함수), 26
- PyErr_SetExcFromWindowsErrWithFilenameObject (C 함수), 25
- PyErr_SetExcFromWindowsErrWithFilenameObjectEx (C 함수), 25
- PyErr_SetExcInfo (C 함수), 29
- PyErr_SetFromErrno (C 함수), 24
- PyErr_SetFromErrnoWithFilename (C 함수), 25
- PyErr_SetFromErrnoWithFilenameObject (C 함수), 25
- PyErr_SetFromErrnoWithFilenameObjects (C 함수), 25
- PyErr_SetFromWindowsErr (C 함수), 25
- PyErr_SetFromWindowsErrWithFilename (C 함수), 25
- PyErr_SetImportError (C 함수), 26
- PyErr_SetImportErrorSubclass (C 함수), 27
- PyErr_SetInterrupt (C 함수), 29
- PyErr_SetNone (C 함수), 24
- PyErr_SetObject (C 함수), 24
- PyErr_SetString (C 함수), 24
- PyErr_SetString(), 9
- PyErr_SyntaxLocation (C 함수), 26
- PyErr_SyntaxLocationEx (C 함수), 26
- PyErr_SyntaxLocationObject (C 함수), 26
- PyErr_WarnEx (C 함수), 26
- PyErr_WarnExplicit (C 함수), 27
- PyErr_WarnExplicitObject (C 함수), 27
- PyErr_WarnFormat (C 함수), 27
- PyErr_WriteUnraisable (C 함수), 24
- PyEval_AcquireLock (C 함수), 151
- PyEval_AcquireThread (C 함수), 151
- PyEval_AcquireThread(), 148
- PyEval_EvalCode (C 함수), 18
- PyEval_EvalCodeEx (C 함수), 18
- PyEval_EvalFrame (C 함수), 19
- PyEval_EvalFrameEx (C 함수), 19
- PyEval_GetBuiltins (C 함수), 54
- PyEval_GetFrame (C 함수), 54
- PyEval_GetFuncDesc (C 함수), 54
- PyEval_GetFuncName (C 함수), 54
- PyEval_GetGlobals (C 함수), 54
- PyEval_GetLocals (C 함수), 54
- PyEval_InitThreads (C 함수), 148
- PyEval_InitThreads(), 142
- PyEval_MergeCompilerFlags (C 함수), 19
- PyEval_ReleaseLock (C 함수), 152
- PyEval_ReleaseThread (C 함수), 151
- PyEval_ReleaseThread(), 148
- PyEval_RestoreThread (C 함수), 148
- PyEval_RestoreThread(), 147, 148
- PyEval_SaveThread (C 함수), 148
- PyEval_SaveThread(), 147, 148
- PyEval_SetProfile (C 함수), 155
- PyEval_SetTrace (C 함수), 155
- PyEval_ThreadsInitialized (C 함수), 148
- PyExc_ArithmeticError, 32
- PyExc_AssertionError, 32
- PyExc_AttributeError, 32
- PyExc_BaseException, 32
- PyExc_BlockingIOError, 32
- PyExc_BrokenPipeError, 32
- PyExc_BufferError, 32
- PyExc_BytesWarning, 34
- PyExc_ChildProcessError, 32
- PyExc_ConnectionAbortedError, 32
- PyExc_ConnectionError, 32
- PyExc_ConnectionRefusedError, 32
- PyExc_ConnectionResetError, 32
- PyExc_DeprecationWarning, 34
- PyExc_EnvironmentError, 34
- PyExc_EOFError, 32
- PyExc_Exception, 32
- PyExc_FileExistsError, 32
- PyExc_FileNotFoundError, 32
- PyExc_FloatingPointError, 32
- PyExc_FutureWarning, 34
- PyExc_GeneratorExit, 32
- PyExc_ImportError, 32
- PyExc_ImportWarning, 34
- PyExc_IndentationError, 32
- PyExc_IndexError, 32
- PyExc_InterruptedError, 32

PyExc_IOError, 34
 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, 34
 PyExc_PermissionError, 32
 PyExc_ProcessLookupError, 32
 PyExc_RecursionError, 32
 PyExc_ReferenceError, 32
 PyExc_ResourceWarning, 34
 PyExc_RuntimeError, 32
 PyExc_RuntimeWarning, 34
 PyExc_StopAsyncIteration, 32
 PyExc_StopIteration, 32
 PyExc_SyntaxError, 32
 PyExc_SyntaxWarning, 34
 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, 34
 PyExc_UserWarning, 34
 PyExc_ValueError, 32
 PyExc_Warning, 34
 PyExc_WindowsError, 34
 PyExc_ZeroDivisionError, 32
 PyException_GetCause (C 함수), 30
 PyException_GetContext (C 함수), 30
 PyException_GetTraceback (C 함수), 30
 PyException_SetCause (C 함수), 30
 PyException_SetContext (C 함수), 30
 PyException_SetTraceback (C 함수), 30
 PyFile_FromFd (C 함수), 120
 PyFile_GetLine (C 함수), 120
 PyFile_SetOpenCodeHook (C 함수), 121
 PyFile_WriteObject (C 함수), 121
 PyFile_WriteString (C 함수), 121
 PyFloat_AS_DOUBLE (C 함수), 84
 PyFloat_AsDouble (C 함수), 84
 PyFloat_Check (C 함수), 84
 PyFloat_CheckExact (C 함수), 84
 PyFloat_ClearFreeList (C 함수), 84
 PyFloat_FromDouble (C 함수), 84
 PyFloat_FromString (C 함수), 84
 PyFloat_GetInfo (C 함수), 84
 PyFloat_GetMax (C 함수), 84
 PyFloat_GetMin (C 함수), 84
 PyFloat_Type (C 변수), 84
 PyFloatObject (C 데이터 형식), 84
 PyFrame_GetLineNumber (C 함수), 54
 PyFrameObject (C 데이터 형식), 19
 PyFrozenSet_Check (C 함수), 115
 PyFrozenSet_CheckExact (C 함수), 115
 PyFrozenSet_New (C 함수), 116
 PyFrozenSet_Type (C 변수), 115
 PyFunction_Check (C 함수), 117
 PyFunction_GetAnnotations (C 함수), 117
 PyFunction_GetClosure (C 함수), 117
 PyFunction_GetCode (C 함수), 117
 PyFunction_GetDefaults (C 함수), 117
 PyFunction_GetGlobals (C 함수), 117
 PyFunction_GetModule (C 함수), 117
 PyFunction_New (C 함수), 117
 PyFunction_NewWithQualName (C 함수), 117
 PyFunction_SetAnnotations (C 함수), 118
 PyFunction_SetClosure (C 함수), 117
 PyFunction_SetDefaults (C 함수), 117
 PyFunction_Type (C 변수), 117
 PyFunctionObject (C 데이터 형식), 117
 PyGen_Check (C 함수), 133
 PyGen_CheckExact (C 함수), 133
 PyGen_New (C 함수), 133
 PyGen_NewWithQualName (C 함수), 133
 PyGen_Type (C 변수), 133
 PyGenObject (C 데이터 형식), 133
 PyGetSetDef (C 데이터 형식), 189
 PyGILState_Check (C 함수), 149
 PyGILState_Ensure (C 함수), 149
 PyGILState_GetThisThreadState (C 함수), 149
 PyGILState_Release (C 함수), 149
 PyImport_AddModule (C 함수), 41
 PyImport_AddModuleObject (C 함수), 41
 PyImport_AppendInittab (C 함수), 43
 PyImport_Cleanup (C 함수), 43
 PyImport_ExecCodeModule (C 함수), 41
 PyImport_ExecCodeModuleEx (C 함수), 42
 PyImport_ExecCodeModuleObject (C 함수), 42
 PyImport_ExecCodeModuleWithPathnames (C 함수), 42
 PyImport_ExtendInittab (C 함수), 43
 PyImport_FrozenModules (C 변수), 43
 PyImport_GetImporter (C 함수), 42
 PyImport_GetMagicNumber (C 함수), 42
 PyImport_GetMagicTag (C 함수), 42

- PyImport_GetModule (C 함수), 42
- PyImport_GetModuleDict (C 함수), 42
- PyImport_Import (C 함수), 41
- PyImport_ImportFrozenModule (C 함수), 43
- PyImport_ImportFrozenModuleObject (C 함수), 43
- PyImport_ImportModule (C 함수), 40
- PyImport_ImportModuleEx (C 함수), 40
- PyImport_ImportModuleLevel (C 함수), 41
- PyImport_ImportModuleLevelObject (C 함수), 40
- PyImport_ImportModuleNoBlock (C 함수), 40
- PyImport_ReloadModule (C 함수), 41
- PyIndex_Check (C 함수), 65
- PyInstanceMethod_Check (C 함수), 118
- PyInstanceMethod_Function (C 함수), 118
- PyInstanceMethod_GET_FUNCTION (C 함수), 118
- PyInstanceMethod_New (C 함수), 118
- PyInstanceMethod_Type (C 변수), 118
- PyInterpreterState (C 데이터 형식), 148
- PyInterpreterState_Clear (C 함수), 150
- PyInterpreterState_Delete (C 함수), 150
- PyInterpreterState_GetDict (C 함수), 151
- PyInterpreterState_GetID (C 함수), 150
- PyInterpreterState_Head (C 함수), 155
- PyInterpreterState_Main (C 함수), 155
- PyInterpreterState_New (C 함수), 150
- PyInterpreterState_Next (C 함수), 155
- PyInterpreterState_ThreadHead (C 함수), 155
- PyIter_Check (C 함수), 69
- PyIter_Next (C 함수), 69
- PyList_Append (C 함수), 112
- PyList_AsTuple (C 함수), 112
- PyList_Check (C 함수), 111
- PyList_CheckExact (C 함수), 111
- PyList_ClearFreeList (C 함수), 112
- PyList_GET_ITEM (C 함수), 111
- PyList_GET_SIZE (C 함수), 111
- PyList_GetItem (C 함수), 111
- PyList_GetItem(), 8
- PyList_GetSlice (C 함수), 112
- PyList_Insert (C 함수), 112
- PyList_New (C 함수), 111
- PyList_Reverse (C 함수), 112
- PyList_SET_ITEM (C 함수), 111
- PyList_SetItem (C 함수), 111
- PyList_SetItem(), 7
- PyList_SetSlice (C 함수), 112
- PyList_Size (C 함수), 111
- PyList_Sort (C 함수), 112
- PyList_Type (C 변수), 111
- PyListObject (C 데이터 형식), 111
- PyLong_AsDouble (C 함수), 83
- PyLong_AsLong (C 함수), 81
- PyLong_AsLongAndOverflow (C 함수), 81
- PyLong_AsLongLong (C 함수), 82
- PyLong_AsLongLongAndOverflow (C 함수), 82
- PyLong_AsSize_t (C 함수), 82
- PyLong_AsSsize_t (C 함수), 82
- PyLong_AsUnsignedLong (C 함수), 82
- PyLong_AsUnsignedLongLong (C 함수), 82
- PyLong_AsUnsignedLongLongMask (C 함수), 83
- PyLong_AsUnsignedLongMask (C 함수), 83
- PyLong_AsVoidPtr (C 함수), 83
- PyLong_Check (C 함수), 80
- PyLong_CheckExact (C 함수), 80
- PyLong_FromDouble (C 함수), 81
- PyLong_FromLong (C 함수), 80
- PyLong_FromLongLong (C 함수), 81
- PyLong_FromSize_t (C 함수), 81
- PyLong_FromSsize_t (C 함수), 81
- PyLong_FromString (C 함수), 81
- PyLong_FromUnicode (C 함수), 81
- PyLong_FromUnicodeObject (C 함수), 81
- PyLong_FromUnsignedLong (C 함수), 81
- PyLong_FromUnsignedLongLong (C 함수), 81
- PyLong_FromVoidPtr (C 함수), 81
- PyLong_Type (C 변수), 80
- PyLongObject (C 데이터 형식), 80
- PyMapping_Check (C 함수), 68
- PyMapping_DelItem (C 함수), 68
- PyMapping_DelItemString (C 함수), 68
- PyMapping_GetItemString (C 함수), 68
- PyMapping_HasKey (C 함수), 68
- PyMapping_HasKeyString (C 함수), 68
- PyMapping_Items (C 함수), 68
- PyMapping_Keys (C 함수), 68
- PyMapping_Length (C 함수), 68
- PyMapping_SetItemString (C 함수), 68
- PyMapping_Size (C 함수), 68
- PyMapping_Values (C 함수), 68
- PyMappingMethods (C 데이터 형식), 217
- PyMappingMethods.mp_ass_subscript (C 멤버 변수), 217
- PyMappingMethods.mp_length (C 멤버 변수), 217
- PyMappingMethods.mp_subscript (C 멤버 변수), 217
- PyMarshal_ReadLastObjectFromFile (C 함수), 44
- PyMarshal_ReadLongFromFile (C 함수), 44
- PyMarshal_ReadObjectFromFile (C 함수), 44
- PyMarshal_ReadObjectFromString (C 함수), 44
- PyMarshal_ReadShortFromFile (C 함수), 44
- PyMarshal_WriteLongToFile (C 함수), 44
- PyMarshal_WriteObjectToFile (C 함수), 44
- PyMarshal_WriteObjectToString (C 함수), 44

- PyMem_Calloc (C 함수), 177
- PyMem_Del (C 함수), 178
- PYMEM_DOMAIN_MEM (C 매크로), 180
- PYMEM_DOMAIN_OBJ (C 매크로), 180
- PYMEM_DOMAIN_RAW (C 매크로), 180
- PyMem_Free (C 함수), 177
- PyMem_GetAllocator (C 함수), 180
- PyMem_Malloc (C 함수), 177
- PyMem_New (C 함수), 177
- PyMem_RawCalloc (C 함수), 176
- PyMem_RawFree (C 함수), 177
- PyMem_RawMalloc (C 함수), 176
- PyMem_RawRealloc (C 함수), 176
- PyMem_Realloc (C 함수), 177
- PyMem_Resize (C 함수), 178
- PyMem_SetAllocator (C 함수), 180
- PyMem_SetupDebugHooks (C 함수), 180
- PyMemAllocatorDomain (C 데이터 형식), 180
- PyMemAllocatorEx (C 데이터 형식), 179
- PyMemberDef (C 데이터 형식), 188
- PyMemoryView_Check (C 함수), 130
- PyMemoryView_FromBuffer (C 함수), 130
- PyMemoryView_FromMemory (C 함수), 130
- PyMemoryView_FromObject (C 함수), 130
- PyMemoryView_GET_BASE (C 함수), 130
- PyMemoryView_GET_BUFFER (C 함수), 130
- PyMemoryView_GetContiguous (C 함수), 130
- PyMethod_Check (C 함수), 118
- PyMethod_ClearFreeList (C 함수), 119
- PyMethod_Function (C 함수), 118
- PyMethod_GET_FUNCTION (C 함수), 118
- PyMethod_GET_SELF (C 함수), 118
- PyMethod_New (C 함수), 118
- PyMethod_Self (C 함수), 118
- PyMethod_Type (C 변수), 118
- PyMethodDef (C 데이터 형식), 187
- PyModule_AddFunctions (C 함수), 125
- PyModule_AddIntConstant (C 함수), 126
- PyModule_AddIntMacro (C 함수), 126
- PyModule_AddObject (C 함수), 126
- PyModule_AddStringConstant (C 함수), 126
- PyModule_AddStringMacro (C 함수), 126
- PyModule_Check (C 함수), 121
- PyModule_CheckExact (C 함수), 121
- PyModule_Create (C 함수), 123
- PyModule_Create2 (C 함수), 123
- PyModule_ExecDef (C 함수), 125
- PyModule_FromDefAndSpec (C 함수), 125
- PyModule_FromDefAndSpec2 (C 함수), 125
- PyModule_GetDef (C 함수), 122
- PyModule_GetDict (C 함수), 121
- PyModule_GetFilename (C 함수), 122
- PyModule_GetFilenameObject (C 함수), 122
- PyModule_GetName (C 함수), 122
- PyModule_GetNameObject (C 함수), 122
- PyModule_GetState (C 함수), 122
- PyModule_New (C 함수), 121
- PyModule_NewObject (C 함수), 121
- PyModule_SetDocString (C 함수), 125
- PyModule_Type (C 변수), 121
- PyModuleDef (C 데이터 형식), 122
- PyModuleDef_Init (C 함수), 124
- PyModuleDef_Slot (C 데이터 형식), 124
- PyModuleDef_Slot.slot (C 멤버 변수), 124
- PyModuleDef_Slot.value (C 멤버 변수), 124
- PyModuleDef.m_base (C 멤버 변수), 122
- PyModuleDef.m_clear (C 멤버 변수), 123
- PyModuleDef.m_doc (C 멤버 변수), 122
- PyModuleDef.m_free (C 멤버 변수), 123
- PyModuleDef.m_methods (C 멤버 변수), 123
- PyModuleDef.m_name (C 멤버 변수), 122
- PyModuleDef.m_reload (C 멤버 변수), 123
- PyModuleDef.m_size (C 멤버 변수), 122
- PyModuleDef.m_slots (C 멤버 변수), 123
- PyModuleDef.m_traverse (C 멤버 변수), 123
- PyNumber_Absolute (C 함수), 64
- PyNumber_Add (C 함수), 63
- PyNumber_And (C 함수), 64
- PyNumber_AsSsize_t (C 함수), 65
- PyNumber_Check (C 함수), 63
- PyNumber_Divmod (C 함수), 63
- PyNumber_Float (C 함수), 65
- PyNumber_FloorDivide (C 함수), 63
- PyNumber_Index (C 함수), 65
- PyNumber_InPlaceAdd (C 함수), 64
- PyNumber_InPlaceAnd (C 함수), 65
- PyNumber_InPlaceFloorDivide (C 함수), 64
- PyNumber_InPlaceLshift (C 함수), 65
- PyNumber_InPlaceMatrixMultiply (C 함수), 64
- PyNumber_InPlaceMultiply (C 함수), 64
- PyNumber_InPlaceOr (C 함수), 65
- PyNumber_InPlacePower (C 함수), 65
- PyNumber_InPlaceRemainder (C 함수), 64
- PyNumber_InPlaceRshift (C 함수), 65
- PyNumber_InPlaceSubtract (C 함수), 64
- PyNumber_InPlaceTrueDivide (C 함수), 64
- PyNumber_InPlaceXor (C 함수), 65
- PyNumber_Invert (C 함수), 64
- PyNumber_Long (C 함수), 65
- PyNumber_Lshift (C 함수), 64
- PyNumber_MatrixMultiply (C 함수), 63
- PyNumber_Multiply (C 함수), 63
- PyNumber_Negative (C 함수), 63
- PyNumber_Or (C 함수), 64
- PyNumber_Positive (C 함수), 63
- PyNumber_Power (C 함수), 63
- PyNumber_Remainder (C 함수), 63

- PyNumber_Rshift (C 함수), 64
- PyNumber_Subtract (C 함수), 63
- PyNumber_ToBase (C 함수), 65
- PyNumber_TrueDivide (C 함수), 63
- PyNumber_Xor (C 함수), 64
- PyNumberMethods (C 데이터 형식), 214
- PyNumberMethods.nb_absolute (C 멤버 변수), 216
- PyNumberMethods.nb_add (C 멤버 변수), 215
- PyNumberMethods.nb_and (C 멤버 변수), 216
- PyNumberMethods.nb_bool (C 멤버 변수), 216
- PyNumberMethods.nb_divmod (C 멤버 변수), 216
- PyNumberMethods.nb_float (C 멤버 변수), 216
- PyNumberMethods.nb_floor_divide (C 멤버 변수), 216
- PyNumberMethods.nb_index (C 멤버 변수), 216
- PyNumberMethods.nb_inplace_add (C 멤버 변수), 216
- PyNumberMethods.nb_inplace_and (C 멤버 변수), 216
- PyNumberMethods.nb_inplace_floor_divide (C 멤버 변수), 216
- PyNumberMethods.nb_inplace_lshift (C 멤버 변수), 216
- PyNumberMethods.nb_inplace_matrix_multiply (C 멤버 변수), 216
- PyNumberMethods.nb_inplace_multiply (C 멤버 변수), 216
- PyNumberMethods.nb_inplace_or (C 멤버 변수), 216
- PyNumberMethods.nb_inplace_power (C 멤버 변수), 216
- PyNumberMethods.nb_inplace_remainder (C 멤버 변수), 216
- PyNumberMethods.nb_inplace_rshift (C 멤버 변수), 216
- PyNumberMethods.nb_inplace_subtract (C 멤버 변수), 216
- PyNumberMethods.nb_inplace_true_divide (C 멤버 변수), 216
- PyNumberMethods.nb_inplace_xor (C 멤버 변수), 216
- PyNumberMethods.nb_int (C 멤버 변수), 216
- PyNumberMethods.nb_invert (C 멤버 변수), 216
- PyNumberMethods.nb_lshift (C 멤버 변수), 216
- PyNumberMethods.nb_matrix_multiply (C 멤버 변수), 216
- PyNumberMethods.nb_multiply (C 멤버 변수), 215
- PyNumberMethods.nb_negative (C 멤버 변수), 216
- PyNumberMethods.nb_or (C 멤버 변수), 216
- PyNumberMethods.nb_positive (C 멤버 변수), 216
- PyNumberMethods.nb_power (C 멤버 변수), 216
- PyNumberMethods.nb_remainder (C 멤버 변수), 216
- PyNumberMethods.nb_reserved (C 멤버 변수), 216
- PyNumberMethods.nb_rshift (C 멤버 변수), 216
- PyNumberMethods.nb_subtract (C 멤버 변수), 215
- PyNumberMethods.nb_true_divide (C 멤버 변수), 216
- PyNumberMethods.nb_xor (C 멤버 변수), 216
- PyObject (C 데이터 형식), 186
- PyObject_AsCharBuffer (C 함수), 76
- PyObject_ASCII (C 함수), 59
- PyObject_AsFileDescriptor (C 함수), 120
- PyObject_AsReadBuffer (C 함수), 76
- PyObject_AsWriteBuffer (C 함수), 76
- PyObject_Bytes (C 함수), 59
- PyObject_Call (C 함수), 60
- PyObject_CallFunction (C 함수), 60
- PyObject_CallFunctionObjArgs (C 함수), 60
- PyObject_CallMethod (C 함수), 60
- PyObject_CallMethodObjArgs (C 함수), 60
- PyObject_CallObject (C 함수), 60
- PyObject_Calloc (C 함수), 178
- PyObject_CheckBuffer (C 함수), 75
- PyObject_CheckReadBuffer (C 함수), 76
- PyObject_Del (C 함수), 185
- PyObject_DelAttr (C 함수), 58
- PyObject_DelAttrString (C 함수), 58
- PyObject_DelItem (C 함수), 62
- PyObject_Dir (C 함수), 62
- PyObject_Free (C 함수), 179
- PyObject_GC_Del (C 함수), 224
- PyObject_GC_New (C 함수), 224
- PyObject_GC_NewVar (C 함수), 224
- PyObject_GC_Resize (C 함수), 224
- PyObject_GC_Track (C 함수), 224
- PyObject_GC_UnTrack (C 함수), 224
- PyObject_GenericGetAttr (C 함수), 58
- PyObject_GenericGetDict (C 함수), 58
- PyObject_GenericSetAttr (C 함수), 58
- PyObject_GenericSetDict (C 함수), 58
- PyObject_GetArenaAllocator (C 함수), 182
- PyObject_GetAttr (C 함수), 57
- PyObject_GetAttrString (C 함수), 58
- PyObject_GetBuffer (C 함수), 75
- PyObject_GetItem (C 함수), 62
- PyObject_GetIter (C 함수), 62
- PyObject_HasAttr (C 함수), 57
- PyObject_HasAttrString (C 함수), 57
- PyObject_Hash (C 함수), 61
- PyObject_HashNotImplemented (C 함수), 62
- PyObject_HEAD (C 매크로), 186

- PyObject_HEAD_INIT (C 매크로), 187
- PyObject_Init (C 함수), 185
- PyObject_InitVar (C 함수), 185
- PyObject_IsInstance (C 함수), 59
- PyObject_IsSubclass (C 함수), 59
- PyObject_IsTrue (C 함수), 62
- PyObject_Length (C 함수), 62
- PyObject_LengthHint (C 함수), 62
- PyObject_Malloc (C 함수), 178
- PyObject_New (C 함수), 185
- PyObject_NewVar (C 함수), 185
- PyObject_Not (C 함수), 62
- PyObject._ob_next (C 멤버 변수), 196
- PyObject._ob_prev (C 멤버 변수), 196
- PyObject_Print (C 함수), 57
- PyObject_Realloc (C 함수), 178
- PyObject_Repr (C 함수), 59
- PyObject_RichCompare (C 함수), 58
- PyObject_RichCompareBool (C 함수), 58
- PyObject_SetArenaAllocator (C 함수), 182
- PyObject_SetAttr (C 함수), 58
- PyObject_SetAttrString (C 함수), 58
- PyObject_SetItem (C 함수), 62
- PyObject_Size (C 함수), 62
- PyObject_Str (C 함수), 59
- PyObject_Type (C 함수), 62
- PyObject_TypeCheck (C 함수), 62
- PyObject_VAR_HEAD (C 매크로), 186
- PyObjectArenaAllocator (C 데이터 형식), 182
- PyObject.ob_refcnt (C 멤버 변수), 196
- PyObject.ob_type (C 멤버 변수), 197
- PyOS_AfterFork (C 함수), 36
- PyOS_AfterFork_Child (C 함수), 36
- PyOS_AfterFork_Parent (C 함수), 35
- PyOS_BeforeFork (C 함수), 35
- PyOS_CheckStack (C 함수), 36
- PyOS_double_to_string (C 함수), 53
- PyOS_FSPath (C 함수), 35
- PyOS_getsig (C 함수), 36
- PyOS_InputHook (C 변수), 17
- PyOS_ReadlineFunctionPointer (C 변수), 17
- PyOS_setsig (C 함수), 36
- PyOS_snprintf (C 함수), 52
- PyOS_strncmp (C 함수), 54
- PyOS_string_to_double (C 함수), 53
- PyOS_strnicmp (C 함수), 54
- PyOS_vsnprintf (C 함수), 52
- PyParser_SimpleParseFile (C 함수), 17
- PyParser_SimpleParseFileFlags (C 함수), 17
- PyParser_SimpleParseString (C 함수), 17
- PyParser_SimpleParseStringFlags (C 함수), 17
- PyParser_SimpleParseStringFlagsFilename (C 함수), 17
- PyPreConfig (C 데이터 형식), 162
- PyPreConfig_InitIsolatedConfig (C 함수), 162
- PyPreConfig_InitPythonConfig (C 함수), 162
- PyPreConfig.allocator (C 멤버 변수), 162
- PyPreConfig.coerce_c_locale (C 멤버 변수), 163
- PyPreConfig.coerce_c_locale_warn (C 멤버 변수), 163
- PyPreConfig.configure_locale (C 멤버 변수), 162
- PyPreConfig.dev_mode (C 멤버 변수), 163
- PyPreConfig.isolated (C 멤버 변수), 163
- PyPreConfig.legacy_windows_fs_encoding (C 멤버 변수), 163
- PyPreConfig.parse_argv (C 멤버 변수), 163
- PyPreConfig.use_environment (C 멤버 변수), 163
- PyPreConfig.utf8_mode (C 멤버 변수), 163
- PyProperty_Type (C 변수), 128
- PyRun_AnyFile (C 함수), 15
- PyRun_AnyFileEx (C 함수), 15
- PyRun_AnyFileExFlags (C 함수), 16
- PyRun_AnyFileFlags (C 함수), 15
- PyRun_File (C 함수), 17
- PyRun_FileEx (C 함수), 18
- PyRun_FileExFlags (C 함수), 18
- PyRun_FileFlags (C 함수), 18
- PyRun_InteractiveLoop (C 함수), 16
- PyRun_InteractiveLoopFlags (C 함수), 16
- PyRun_InteractiveOne (C 함수), 16
- PyRun_InteractiveOneFlags (C 함수), 16
- PyRun_SimpleFile (C 함수), 16
- PyRun_SimpleFileEx (C 함수), 16
- PyRun_SimpleFileExFlags (C 함수), 16
- PyRun_SimpleString (C 함수), 16
- PyRun_SimpleStringFlags (C 함수), 16
- PyRun_String (C 함수), 17
- PyRun_StringFlags (C 함수), 17
- PySeqIter_Check (C 함수), 127
- PySeqIter_New (C 함수), 127
- PySeqIter_Type (C 변수), 127
- PySequence_Check (C 함수), 66
- PySequence_Concat (C 함수), 66
- PySequence_Contains (C 함수), 67
- PySequence_Count (C 함수), 66
- PySequence_DelItem (C 함수), 66
- PySequence_DelSlice (C 함수), 66
- PySequence_Fast (C 함수), 67
- PySequence_Fast_GET_ITEM (C 함수), 67
- PySequence_Fast_GET_SIZE (C 함수), 67
- PySequence_Fast_ITEMS (C 함수), 67
- PySequence_GetItem (C 함수), 66
- PySequence_GetItem(), 8

- PySequence_GetSlice (C 함수), 66
- PySequence_Index (C 함수), 67
- PySequence_InPlaceConcat (C 함수), 66
- PySequence_InPlaceRepeat (C 함수), 66
- PySequence_ITEM (C 함수), 67
- PySequence_Length (C 함수), 66
- PySequence_List (C 함수), 67
- PySequence_Repeat (C 함수), 66
- PySequence_SetItem (C 함수), 66
- PySequence_SetSlice (C 함수), 66
- PySequence_Size (C 함수), 66
- PySequence_Tuple (C 함수), 67
- PySequenceMethods (C 데이터 형식), 217
- PySequenceMethods.sq_ass_item (C 멤버 변수), 217
- PySequenceMethods.sq_concat (C 멤버 변수), 217
- PySequenceMethods.sq_contains (C 멤버 변수), 217
- PySequenceMethods.sq_inplace_concat (C 멤버 변수), 217
- PySequenceMethods.sq_inplace_repeat (C 멤버 변수), 218
- PySequenceMethods.sq_item (C 멤버 변수), 217
- PySequenceMethods.sq_length (C 멤버 변수), 217
- PySequenceMethods.sq_repeat (C 멤버 변수), 217
- PySet_Add (C 함수), 116
- PySet_Check (C 함수), 115
- PySet_Clear (C 함수), 116
- PySet_ClearFreeList (C 함수), 116
- PySet_Contains (C 함수), 116
- PySet_Discard (C 함수), 116
- PySet_GET_SIZE (C 함수), 116
- PySet_New (C 함수), 116
- PySet_Pop (C 함수), 116
- PySet_Size (C 함수), 116
- PySet_Type (C 변수), 115
- PySetObject (C 데이터 형식), 115
- PySignal_SetWakeupFd (C 함수), 29
- PySlice_AdjustIndices (C 함수), 129
- PySlice_Check (C 함수), 128
- PySlice_GetIndices (C 함수), 128
- PySlice_GetIndicesEx (C 함수), 128
- PySlice_New (C 함수), 128
- PySlice_Type (C 변수), 128
- PySlice_Unpack (C 함수), 129
- PyState_AddModule (C 함수), 127
- PyState_FindModule (C 함수), 127
- PyState_RemoveModule (C 함수), 127
- PyStatus (C 데이터 형식), 161
- PyStatus_Error (C 함수), 161
- PyStatus_Exception (C 함수), 161
- PyStatus_Exit (C 함수), 161
- PyStatus_IsError (C 함수), 161
- PyStatus_IsExit (C 함수), 161
- PyStatus_NoMemory (C 함수), 161
- PyStatus_Ok (C 함수), 161
- PyStatus.err_msg (C 멤버 변수), 161
- PyStatus.exitcode (C 멤버 변수), 161
- PyStatus.func (C 멤버 변수), 161
- PyStructSequence_Desc (C 데이터 형식), 110
- PyStructSequence_Field (C 데이터 형식), 110
- PyStructSequence_GET_ITEM (C 함수), 110
- PyStructSequence_GetItem (C 함수), 110
- PyStructSequence_InitType (C 함수), 110
- PyStructSequence_InitType2 (C 함수), 110
- PyStructSequence_New (C 함수), 110
- PyStructSequence_NewType (C 함수), 110
- PyStructSequence_SET_ITEM (C 함수), 111
- PyStructSequence_SetItem (C 함수), 110
- PyStructSequence_UnnamedField (C 변수), 110
- PySys_AddAuditHook (C 함수), 39
- PySys_AddWarnOption (C 함수), 38
- PySys_AddWarnOptionUnicode (C 함수), 38
- PySys_AddXOption (C 함수), 38
- PySys_Audit (C 함수), 39
- PySys_FormatStderr (C 함수), 38
- PySys_FormatStdout (C 함수), 38
- PySys_GetObject (C 함수), 38
- PySys_GetXOptions (C 함수), 39
- PySys_ResetWarnOptions (C 함수), 38
- PySys_SetArgv (C 함수), 145
- PySys_SetArgv (), 142
- PySys_SetArgvEx (C 함수), 145
- PySys_SetArgvEx (), 11, 142
- PySys_SetObject (C 함수), 38
- PySys_SetPath (C 함수), 38
- PySys_WriteStderr (C 함수), 38
- PySys_WriteStdout (C 함수), 38
- Python 3000 (파이썬 3000), 239
- PYTHON*, 141
- PYTHONCOERCECLOCALE, 170
- PYTHONDEBUG, 140
- PYTHONDONTWRITEBYTECODE, 140
- PYTHONDUMPREFS, 196
- PYTHONHASHSEED, 141
- PYTHONHOME, 11, 12, 141, 146, 166
- Pythonic (파이썬다운), 239
- PYTHONINSPECT, 141
- PYTHONIOENCODING, 143
- PYTHONLEGACYWINDOWSFSENCODING, 141
- PYTHONLEGACYWINDOWSSTDIO, 141
- PYTHONMALLOC, 176, 179, 181
- PYTHONMALLOCSTATS, 176
- PYTHONNOUSERSITE, 141
- PYTHONOPTIMIZE, 141

PYTHONPATH, 11, 12, 141, 167
 PYTHONUNBUFFERED, 142
 PYTHONUTF8, 170
 PYTHONVERBOSE, 142
 PyThread_create_key (C 함수), 157
 PyThread_delete_key (C 함수), 157
 PyThread_delete_key_value (C 함수), 157
 PyThread_get_key_value (C 함수), 157
 PyThread_ReInitTLS (C 함수), 157
 PyThread_set_key_value (C 함수), 157
 PyThread_tss_alloc (C 함수), 156
 PyThread_tss_create (C 함수), 157
 PyThread_tss_delete (C 함수), 157
 PyThread_tss_free (C 함수), 156
 PyThread_tss_get (C 함수), 157
 PyThread_tss_is_created (C 함수), 157
 PyThread_tss_set (C 함수), 157
 PyThreadState, 146
 PyThreadState (C 데이터 형식), 148
 PyThreadState_Clear (C 함수), 150
 PyThreadState_Delete (C 함수), 150
 PyThreadState_Get (C 함수), 149
 PyThreadState_GetDict (C 함수), 151
 PyThreadState_New (C 함수), 150
 PyThreadState_Next (C 함수), 155
 PyThreadState_SetAsyncExc (C 함수), 151
 PyThreadState_Swap (C 함수), 149
 PyTime_Check (C 함수), 136
 PyTime_CheckExact (C 함수), 136
 PyTime_FromTime (C 함수), 136
 PyTime_FromTimeAndFold (C 함수), 136
 PyTimeZone_FromOffset (C 함수), 137
 PyTimeZone_FromOffsetAndName (C 함수), 137
 PyTrace_C_CALL (C 변수), 154
 PyTrace_C_EXCEPTION (C 변수), 155
 PyTrace_C_RETURN (C 변수), 155
 PyTrace_CALL (C 변수), 154
 PyTrace_EXCEPTION (C 변수), 154
 PyTrace_LINE (C 변수), 154
 PyTrace_OPCODE (C 변수), 155
 PyTrace_RETURN (C 변수), 154
 PyTraceMalloc_Track (C 함수), 182
 PyTraceMalloc_Untrack (C 함수), 182
 PyTuple_Check (C 함수), 108
 PyTuple_CheckExact (C 함수), 108
 PyTuple_ClearFreeList (C 함수), 109
 PyTuple_GET_ITEM (C 함수), 109
 PyTuple_GET_SIZE (C 함수), 109
 PyTuple_GetItem (C 함수), 109
 PyTuple_GetSlice (C 함수), 109
 PyTuple_New (C 함수), 108
 PyTuple_Pack (C 함수), 108
 PyTuple_SET_ITEM (C 함수), 109
 PyTuple_SetItem (C 함수), 109
 PyTuple_SetItem(), 7
 PyTuple_Size (C 함수), 108
 PyTuple_Type (C 변수), 108
 PyTupleObject (C 데이터 형식), 108
 PyType_Check (C 함수), 77
 PyType_CheckExact (C 함수), 77
 PyType_ClearCache (C 함수), 77
 PyType_FromSpec (C 함수), 79
 PyType_FromSpecWithBases (C 함수), 79
 PyType_GenericAlloc (C 함수), 78
 PyType_GenericNew (C 함수), 78
 PyType_GetFlags (C 함수), 78
 PyType_GetSlot (C 함수), 78
 PyType_HasFeature (C 함수), 78
 PyType_IS_GC (C 함수), 78
 PyType_IsSubtype (C 함수), 78
 PyType_Modified (C 함수), 78
 PyType_Ready (C 함수), 78
 PyType_Slot (C 데이터 형식), 79
 PyType_Slot.PyType_Slot.pfunc (C 멤버 변수), 80
 PyType_Slot.PyType_Slot.slot (C 멤버 변수), 79
 PyType_Spec (C 데이터 형식), 79
 PyType_Spec.PyType_Spec.basicsize (C 멤버 변수), 79
 PyType_Spec.PyType_Spec.flags (C 멤버 변수), 79
 PyType_Spec.PyType_Spec.itemsize (C 멤버 변수), 79
 PyType_Spec.PyType_Spec.name (C 멤버 변수), 79
 PyType_Spec.PyType_Spec.slots (C 멤버 변수), 79
 PyType_Type (C 변수), 77
 PyTypeObject (C 데이터 형식), 77
 PyTypeObject.tp_alloc (C 멤버 변수), 211
 PyTypeObject.tp_allocs (C 멤버 변수), 214
 PyTypeObject.tp_as_async (C 멤버 변수), 200
 PyTypeObject.tp_as_buffer (C 멤버 변수), 202
 PyTypeObject.tp_as_mapping (C 멤버 변수), 201
 PyTypeObject.tp_as_number (C 멤버 변수), 200
 PyTypeObject.tp_as_sequence (C 멤버 변수), 200
 PyTypeObject.tp_base (C 멤버 변수), 208
 PyTypeObject.tp_bases (C 멤버 변수), 212
 PyTypeObject.tp_basicsize (C 멤버 변수), 198
 PyTypeObject.tp_cache (C 멤버 변수), 212
 PyTypeObject.tp_call (C 멤버 변수), 201
 PyTypeObject.tp_clear (C 멤버 변수), 206
 PyTypeObject.tp_dealloc (C 멤버 변수), 198
 PyTypeObject.tp_del (C 멤버 변수), 213
 PyTypeObject.tp_descr_get (C 멤버 변수), 209

- PyTypeObject.tp_descr_set (C 멤버 변수), 209
 PyTypeObject.tp_dict (C 멤버 변수), 209
 PyTypeObject.tp_dictoffset (C 멤버 변수), 210
 PyTypeObject.tp_doc (C 멤버 변수), 205
 PyTypeObject.tp_finalize (C 멤버 변수), 213
 PyTypeObject.tp_flags (C 멤버 변수), 202
 PyTypeObject.tp_free (C 멤버 변수), 212
 PyTypeObject.tp_frees (C 멤버 변수), 214
 PyTypeObject.tp_getattr (C 멤버 변수), 199
 PyTypeObject.tp_getattro (C 멤버 변수), 202
 PyTypeObject.tp_getset (C 멤버 변수), 208
 PyTypeObject.tp_hash (C 멤버 변수), 201
 PyTypeObject.tp_init (C 멤버 변수), 210
 PyTypeObject.tp_is_gc (C 멤버 변수), 212
 PyTypeObject.tp_ismember (C 멤버 변수), 198
 PyTypeObject.tp_iter (C 멤버 변수), 208
 PyTypeObject.tp_iternext (C 멤버 변수), 208
 PyTypeObject.tp_maxalloc (C 멤버 변수), 214
 PyTypeObject.tp_members (C 멤버 변수), 208
 PyTypeObject.tp_methods (C 멤버 변수), 208
 PyTypeObject.tp_mro (C 멤버 변수), 212
 PyTypeObject.tp_name (C 멤버 변수), 197
 PyTypeObject.tp_new (C 멤버 변수), 211
 PyTypeObject.tp_next (C 멤버 변수), 214
 PyTypeObject.tp_prev (C 멤버 변수), 214
 PyTypeObject.tp_repr (C 멤버 변수), 200
 PyTypeObject.tp_richcompare (C 멤버 변수), 206
 PyTypeObject.tp_setattr (C 멤버 변수), 200
 PyTypeObject.tp_setattro (C 멤버 변수), 202
 PyTypeObject.tp_str (C 멤버 변수), 201
 PyTypeObject.tp_subclasses (C 멤버 변수), 213
 PyTypeObject.tp_traverse (C 멤버 변수), 205
 PyTypeObject.tp_vectorcall_offset (C 멤버 변수), 199
 PyTypeObject.tp_version_tag (C 멤버 변수), 213
 PyTypeObject.tp_weaklist (C 멤버 변수), 213
 PyTypeObject.tp_weaklistoffset (C 멤버 변수), 207
 PyTZInfo_Check (C 함수), 136
 PyTZInfo_CheckExact (C 함수), 136
 PyUnicode_1BYTE_DATA (C 함수), 90
 PyUnicode_1BYTE_KIND (C 매크로), 90
 PyUnicode_2BYTE_DATA (C 함수), 90
 PyUnicode_2BYTE_KIND (C 매크로), 90
 PyUnicode_4BYTE_DATA (C 함수), 90
 PyUnicode_4BYTE_KIND (C 매크로), 90
 PyUnicode_AS_DATA (C 함수), 91
 PyUnicode_AS_UNICODE (C 함수), 91
 PyUnicode_AsASCIIString (C 함수), 105
 PyUnicode_AsCharmapString (C 함수), 105
 PyUnicode_AsEncodedString (C 함수), 100
 PyUnicode_AsLatin1String (C 함수), 104
 PyUnicode_AsMBCSString (C 함수), 106
 PyUnicode_AsRawUnicodeEscapeString (C 함수), 104
 PyUnicode_AsUCS4 (C 함수), 95
 PyUnicode_AsUCS4Copy (C 함수), 96
 PyUnicode_AsUnicode (C 함수), 96
 PyUnicode_AsUnicodeAndSize (C 함수), 96
 PyUnicode_AsUnicodeCopy (C 함수), 97
 PyUnicode_AsUnicodeEscapeString (C 함수), 104
 PyUnicode_AsUTF8 (C 함수), 101
 PyUnicode_AsUTF8AndSize (C 함수), 101
 PyUnicode_AsUTF8String (C 함수), 100
 PyUnicode_AsUTF16String (C 함수), 102
 PyUnicode_AsUTF32String (C 함수), 102
 PyUnicode_AsWideChar (C 함수), 99
 PyUnicode_AsWideCharString (C 함수), 99
 PyUnicode_Check (C 함수), 90
 PyUnicode_CheckExact (C 함수), 90
 PyUnicode_ClearFreeList (C 함수), 91
 PyUnicode_Compare (C 함수), 107
 PyUnicode_CompareWithASCIIString (C 함수), 108
 PyUnicode_Concat (C 함수), 107
 PyUnicode_Contains (C 함수), 108
 PyUnicode_CopyCharacters (C 함수), 95
 PyUnicode_Count (C 함수), 107
 PyUnicode_DATA (C 함수), 91
 PyUnicode_Decode (C 함수), 100
 PyUnicode_DecodeASCII (C 함수), 105
 PyUnicode_DecodeCharmap (C 함수), 105
 PyUnicode_DecodeFSDefault (C 함수), 98
 PyUnicode_DecodeFSDefaultAndSize (C 함수), 98
 PyUnicode_DecodeLatin1 (C 함수), 104
 PyUnicode_DecodeLocale (C 함수), 97
 PyUnicode_DecodeLocaleAndSize (C 함수), 97
 PyUnicode_DecodeMBCS (C 함수), 106
 PyUnicode_DecodeMBCSStateful (C 함수), 106
 PyUnicode_DecodeRawUnicodeEscape (C 함수), 104
 PyUnicode_DecodeUnicodeEscape (C 함수), 104
 PyUnicode_DecodeUTF7 (C 함수), 103
 PyUnicode_DecodeUTF7Stateful (C 함수), 103
 PyUnicode_DecodeUTF8 (C 함수), 100
 PyUnicode_DecodeUTF8Stateful (C 함수), 100
 PyUnicode_DecodeUTF16 (C 함수), 102
 PyUnicode_DecodeUTF16Stateful (C 함수), 102
 PyUnicode_DecodeUTF32 (C 함수), 101
 PyUnicode_DecodeUTF32Stateful (C 함수), 101
 PyUnicode_Encode (C 함수), 100
 PyUnicode_EncodeASCII (C 함수), 105

- PyUnicode_EncodeCharmap (C 함수), 105
- PyUnicode_EncodeCodePage (C 함수), 106
- PyUnicode_EncodeFSDefault (C 함수), 99
- PyUnicode_EncodeLatin1 (C 함수), 104
- PyUnicode_EncodeLocale (C 함수), 97
- PyUnicode_EncodeMBCS (C 함수), 106
- PyUnicode_EncodeRawUnicodeEscape (C 함수), 104
- PyUnicode_EncodeUnicodeEscape (C 함수), 104
- PyUnicode_EncodeUTF7 (C 함수), 103
- PyUnicode_EncodeUTF8 (C 함수), 101
- PyUnicode_EncodeUTF16 (C 함수), 103
- PyUnicode_EncodeUTF32 (C 함수), 102
- PyUnicode_Fill (C 함수), 95
- PyUnicode_Find (C 함수), 107
- PyUnicode_FindChar (C 함수), 107
- PyUnicode_Format (C 함수), 108
- PyUnicode_FromEncodedObject (C 함수), 95
- PyUnicode_FromFormat (C 함수), 94
- PyUnicode_FromFormatV (C 함수), 95
- PyUnicode_FromKindAndData (C 함수), 93
- PyUnicode_FromObject (C 함수), 97
- PyUnicode_FromString (C 함수), 94
- PyUnicode_FromString(), 113
- PyUnicode_FromStringAndSize (C 함수), 93
- PyUnicode_FromUnicode (C 함수), 96
- PyUnicode_FromWideChar (C 함수), 99
- PyUnicode_FSConverter (C 함수), 98
- PyUnicode_FSDecoder (C 함수), 98
- PyUnicode_GET_DATA_SIZE (C 함수), 91
- PyUnicode_GET_LENGTH (C 함수), 90
- PyUnicode_GET_SIZE (C 함수), 91
- PyUnicode_GetLength (C 함수), 95
- PyUnicode_GetSize (C 함수), 97
- PyUnicode_InternFromString (C 함수), 108
- PyUnicode_InternInPlace (C 함수), 108
- PyUnicode_Join (C 함수), 107
- PyUnicode_KIND (C 함수), 91
- PyUnicode_MAX_CHAR_VALUE (C 매크로), 91
- PyUnicode_New (C 함수), 93
- PyUnicode_READ (C 함수), 91
- PyUnicode_READ_CHAR (C 함수), 91
- PyUnicode_ReadChar (C 함수), 95
- PyUnicode_READY (C 함수), 90
- PyUnicode_Replace (C 함수), 107
- PyUnicode_RichCompare (C 함수), 108
- PyUnicode_Split (C 함수), 107
- PyUnicode_Splitlines (C 함수), 107
- PyUnicode_Substring (C 함수), 95
- PyUnicode_Tailmatch (C 함수), 107
- PyUnicode_TransformDecimalToASCII (C 함수), 96
- PyUnicode_Translate (C 함수), 106
- PyUnicode_TranslateCharmap (C 함수), 106
- PyUnicode_Type (C 변수), 90
- PyUnicode_WCHAR_KIND (C 매크로), 90
- PyUnicode_WRITE (C 함수), 91
- PyUnicode_WriteChar (C 함수), 95
- PyUnicodeDecodeError_Create (C 함수), 30
- PyUnicodeDecodeError_GetEncoding (C 함수), 31
- PyUnicodeDecodeError_GetEnd (C 함수), 31
- PyUnicodeDecodeError_GetObject (C 함수), 31
- PyUnicodeDecodeError_GetReason (C 함수), 31
- PyUnicodeDecodeError_GetStart (C 함수), 31
- PyUnicodeDecodeError_SetEnd (C 함수), 31
- PyUnicodeDecodeError_SetReason (C 함수), 31
- PyUnicodeDecodeError_SetStart (C 함수), 31
- PyUnicodeEncodeError_Create (C 함수), 30
- PyUnicodeEncodeError_GetEncoding (C 함수), 31
- PyUnicodeEncodeError_GetEnd (C 함수), 31
- PyUnicodeEncodeError_GetObject (C 함수), 31
- PyUnicodeEncodeError_GetReason (C 함수), 31
- PyUnicodeEncodeError_GetStart (C 함수), 31
- PyUnicodeEncodeError_SetEnd (C 함수), 31
- PyUnicodeEncodeError_SetReason (C 함수), 31
- PyUnicodeEncodeError_SetStart (C 함수), 31
- PyUnicodeObject (C 데이터 형식), 90
- PyUnicodeTranslateError_Create (C 함수), 30
- PyUnicodeTranslateError_GetEnd (C 함수), 31
- PyUnicodeTranslateError_GetObject (C 함수), 31
- PyUnicodeTranslateError_GetReason (C 함수), 31
- PyUnicodeTranslateError_GetStart (C 함수), 31
- PyUnicodeTranslateError_SetEnd (C 함수), 31
- PyUnicodeTranslateError_SetReason (C 함수), 31
- PyUnicodeTranslateError_SetStart (C 함수), 31
- PyVarObject (C 데이터 형식), 186
- PyVarObject_HEAD_INIT (C 매크로), 187
- PyVarObject.ob_size (C 멤버 변수), 197
- PyVectorcall_Call (C 함수), 199
- PyVectorcall_NARGS (C 함수), 61
- PyWeakref_Check (C 함수), 130
- PyWeakref_CheckProxy (C 함수), 130

PyWeakref_CheckRef (C 함수), 130
 PyWeakref_GET_OBJECT (C 함수), 131
 PyWeakref_GetObject (C 함수), 131
 PyWeakref_NewProxy (C 함수), 131
 PyWeakref_NewRef (C 함수), 131
 PyWideStringList (C 데이터 형식), 160
 PyWideStringList_Append (C 함수), 160
 PyWideStringList_Insert (C 함수), 160
 PyWideStringList.items (C 멤버 변수), 160
 PyWideStringList.length (C 멤버 변수), 160
 PyWrapper_New (C 함수), 128

Q

qualified name (정규화된 이름), 239

R

realloc(), 175
 reference count (참조 횟수), 239
 regular package (정규 패키지), 239
 releasebufferproc (C 데이터 형식), 221
 repr
 내장 함수, 59, 200
 reprfunc (C 데이터 형식), 220
 richcmpfunc (C 데이터 형식), 221

S

stderr
 stdin stdout, 143
 search
 path, module, 11, 142, 144
 sequence
 객체, 86
 sequence (시퀀스), 240
 set
 객체, 115
 set comprehension, 240
 set_all(), 8
 setattrfunc (C 데이터 형식), 220
 setattrfunc (C 데이터 형식), 220
 setswitchinterval() (in module sys), 146
 SIGINT, 29
 signal
 모듈, 29
 single dispatch (싱글 디스패치), 240
 SIZE_MAX, 82
 slice (슬라이스), 240
 special
 method, 240
 special method (특수 메서드), 240
 ssizeargfunc (C 데이터 형식), 221
 ssizeobjargproc (C 데이터 형식), 221
 statement (문장), 240
 staticmethod
 내장 함수, 188

stderr (in module sys), 152
 stdin
 stdout stderr, 143
 stdin (in module sys), 152
 stdout
 stderr, stdin, 143
 stdout (in module sys), 152
 strerror(), 25
 string
 PyObject_Str (C function), 59
 sum_list(), 9
 sum_sequence(), 9, 10
 sys
 모듈, 11, 142, 152
 SystemError (built-in exception), 122

T

ternaryfunc (C 데이터 형식), 221
 text encoding (텍스트 인코딩), 240
 text file (텍스트 파일), 240
 traverseproc (C 데이터 형식), 225
 triple-quoted string (삼중 따옴표 된 문자열), 240
 tuple
 객체, 108
 내장 함수, 67, 112
 type
 객체, 6, 77
 내장 함수, 62
 type (형), 240
 type alias (형 에일리어스), 240
 type hint (형 힌트), 241

U

ULONG_MAX, 82
 unaryfunc (C 데이터 형식), 221
 universal newlines (유니버설 줄 넘김), 241

V

variable annotation (변수 어노테이션), 241
 vectorcallfunc (C 데이터 형식), 220
 version (in module sys), 144, 145
 virtual environment (가상 환경), 241
 virtual machine (가상 기계), 241
 visitproc (C 데이터 형식), 224

X

내장 함수
 __import__, 40
 abs, 64
 ascii, 59
 bytes, 59
 classmethod, 188

- compile, 41
- divmod, 63
- float, 65
- hash, 62, 201
- int, 65
- len, 62, 66, 68, 111, 114, 116
- pow, 63, 65
- repr, 59, 200
- staticmethod, 188
- tuple, 67, 112
- type, 62

모듈

- __main__, 11, 142, 152
- _thread, 148
- builtins, 11, 142, 152
- signal, 29
- sys, 11, 142, 152

Y

파이썬 향상 제안

- PEP 1, 238
- PEP 7, 3, 5, 6
- PEP 238, 19, 233
- PEP 278, 241
- PEP 302, 233, 236
- PEP 343, 231
- PEP 362, 230, 238
- PEP 383, 97, 98
- PEP 384, 13
- PEP 393, 89, 96
- PEP 411, 239
- PEP 420, 233, 237, 238
- PEP 432, 173
- PEP 442, 214
- PEP 443, 234
- PEP 451, 124, 233
- PEP 484, 229, 233, 241
- PEP 489, 125
- PEP 492, 230, 231
- PEP 498, 232
- PEP 519, 238
- PEP 525, 230
- PEP 526, 229, 241
- PEP 528, 141
- PEP 529, 98, 141
- PEP 538, 170
- PEP 539, 156
- PEP 540, 170
- PEP 552, 165
- PEP 578, 39
- PEP 587, 160
- PEP 623, 89
- PEP 3116, 241
- PEP 3119, 59

- PEP 3121, 123
- PEP 3147, 42
- PEP 3151, 34
- PEP 3155, 239

환경 변수

- exec_prefix, 4
- PATH, 11
- prefix, 4
- PYTHON*, 141
- PYTHONCOERCECLOCALE, 170
- PYTHONDEBUG, 140
- PYTHONDONTWRITEBYTECODE, 140
- PYTHONDUMPPREFS, 196
- PYTHONHASHSEED, 141
- PYTHONHOME, 11, 12, 141, 146, 166
- PYTHONINSPECT, 141
- PYTHONIOENCODING, 143
- PYTHONLEGACYWINDOWSFSENCODING, 141
- PYTHONLEGACYWINDOWSSTDIO, 141
- PYTHONMALLOC, 176, 179, 181
- PYTHONMALLOCSTATS, 176
- PYTHONNOUSERSITE, 141
- PYTHONOPTIMIZE, 141
- PYTHONPATH, 11, 12, 141, 167
- PYTHONUNBUFFERED, 142
- PYTHONUTF8, 170
- PYTHONVERBOSE, 142

Z

Zen of Python (파이썬 젠), 241