The Python/C API

출시 버전 *3.7.17*

Guido van Rossum
and the Python development team

6월 28, 2023

Contents

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 *Parsing arguments and building values* for a description of this macro.

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

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, e.g. `PyObject* func(PyObject *Py_UNUSED(ignored))`.

버전 3.4에 추가.

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 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 possible 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;
    }
    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);
    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/python`*X.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/python`*X.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.

CHAPTER 2

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.dll과 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>=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 붙이기* 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.

CHAPTER 3

The Very High Level Layer

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

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

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

int **`Py_Main`**(int *argc*, wchar_t ****argv*)
: The main program for the standard interpreter. This is made available for programs which embed Python. The *argc* and *argv* parameters should be prepared exactly as those which are passed to a C program's `main()` function (converted to wchar_t according to the user's locale). It is important to note that the argument list may be modified (but the contents of the strings pointed to by the argument list are not). The return value will be `0` if the interpreter exits normally (i.e., without an exception), `1` if the interpreter exits due to an exception, or `2` if the parameter list does not represent a valid Python command line.

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

int **`PyRun_AnyFile`**(FILE **fp*, const char **filename*)
: This is a simplified interface to *`PyRun_AnyFileExFlags()`* below, leaving *closeit* set to `0` and *flags* set to `NULL`.

int **`PyRun_AnyFileFlags`**(FILE **fp*, const char **filename*, *PyCompilerFlags* **flags*)
: This is a simplified interface to *`PyRun_AnyFileExFlags()`* below, leaving the *closeit* argument set to `0`.

int **`PyRun_AnyFileEx`**(FILE **fp*, const char **filename*, int *closeit*)
: This is a simplified interface to *`PyRun_AnyFileExFlags()`* below, leaving the *flags* argument set to `NULL`.

int **`PyRun_AnyFileExFlags`**(FILE **fp*, const char **filename*, int *closeit*, *PyCompilerFlags* **flags*)
: If *fp* refers to a file associated with an interactive device (console or terminal input or Unix pseudo-terminal), return the value of *`PyRun_InteractiveLoop()`*, otherwise return the result of *`PyRun_SimpleFile()`*. *filename* is decoded from the filesystem encoding (`sys.getfilesystemencoding()`). If *filename* is `NULL`, this function uses `"???"` as the filename.

int **PyRun_SimpleString**(const char *command)
This is a simplified interface to *PyRun_SimpleStringFlags()* below, leaving the *PyCompilerFlags** argument set to NULL.

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

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

int **PyRun_SimpleFile**(FILE *fp, const char *filename)
This is a simplified interface to *PyRun_SimpleFileExFlags()* below, leaving *closeit* set to 0 and *flags* set to NULL.

int **PyRun_SimpleFileEx**(FILE *fp, const char *filename, int *closeit*)
This is a simplified interface to *PyRun_SimpleFileExFlags()* below, leaving *flags* set to NULL.

int **PyRun_SimpleFileExFlags**(FILE *fp, const char *filename, int *closeit*, *PyCompilerFlags* *flags)
Similar to *PyRun_SimpleStringFlags()*, but the Python source code is read from *fp* instead of an in-memory string. *filename* should be the name of the file, it is decoded from the filesystem encoding (sys.getfilesystemencoding()). If *closeit* is true, the file is closed before PyRun_SimpleFileExFlags returns.

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

---

int **PyRun_InteractiveOne**(FILE *fp, const char *filename)
This is a simplified interface to *PyRun_InteractiveOneFlags()* below, leaving *flags* set to NULL.

int **PyRun_InteractiveOneFlags**(FILE *fp, const char *filename, *PyCompilerFlags* *flags)
Read and execute a single statement from a file associated with an interactive device according to the *flags* argument. The user will be prompted using sys.ps1 and sys.ps2. *filename* is decoded from the filesystem encoding (sys.getfilesystemencoding()).

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

int **PyRun_InteractiveLoop**(FILE *fp, const char *filename)
This is a simplified interface to *PyRun_InteractiveLoopFlags()* below, leaving *flags* set to NULL.

int **PyRun_InteractiveLoopFlags**(FILE *fp, const char *filename, *PyCompilerFlags* *flags)
Read and execute statements from a file associated with an interactive device until EOF is reached. The user will be prompted using sys.ps1 and sys.ps2. *filename* is decoded from the filesystem encoding (sys.getfilesystemencoding()). Returns 0 at EOF or a negative number upon failure.

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

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

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

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

struct _node* **PyParser_SimpleParseString**(const char *str, int start)
This is a simplified interface to `PyParser_SimpleParseStringFlagsFilename()` below, leaving *filename* set to `NULL` and *flags* set to `0`.

struct _node* **PyParser_SimpleParseStringFlags**(const char *str, int start, int flags)
This is a simplified interface to `PyParser_SimpleParseStringFlagsFilename()` below, leaving *filename* set to `NULL`.

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

struct _node* **PyParser_SimpleParseFile**(FILE *fp, const char *filename, int start)
This is a simplified interface to `PyParser_SimpleParseFileFlags()` below, leaving *flags* set to `0`.

struct _node* **PyParser_SimpleParseFileFlags**(FILE *fp, const char *filename, int start, int flags)
Similar to `PyParser_SimpleParseStringFlagsFilename()`, but the Python source code is read from *fp* instead of an in-memory string.

*PyObject** **PyRun_String**(const char *str, int start, *PyObject* *globals, *PyObject* *locals)
Return value: New reference. This is a simplified interface to `PyRun_StringFlags()` below, leaving *flags* set to `NULL`.

*PyObject** **PyRun_StringFlags**(const char *str, int start, *PyObject* *globals, *PyObject* *locals, *PyCompilerFlags* *flags)
Return value: New reference. Execute Python source code from *str* in the context specified by the objects *globals* and *locals* with the compiler flags specified by *flags*. *globals* must be a dictionary; *locals* can be any object that implements the mapping protocol. The parameter *start* specifies the start token that should be used to parse the source code.

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

*PyObject** **PyRun_File**(FILE *fp, const char *filename, int start, *PyObject* *globals, *PyObject* *locals)
Return value: New reference. This is a simplified interface to `PyRun_FileExFlags()` below, leaving *closeit* set to `0` and *flags* set to `NULL`.

*PyObject** **PyRun_FileEx**(FILE *fp, const char *filename, int start, *PyObject* *globals, *PyObject* *locals, int closeit)
Return value: New reference. This is a simplified interface to `PyRun_FileExFlags()` below, leaving *flags* set to `NULL`.

*PyObject** **PyRun_FileFlags**(FILE *fp, const char *filename, int start, *PyObject* *globals, *PyObject* *locals, *PyCompilerFlags* *flags)
Return value: New reference. This is a simplified interface to `PyRun_FileExFlags()` below, leaving *closeit* set to `0`.

*PyObject** **PyRun_FileExFlags**(FILE *fp, const char *filename, int start, *PyObject* *globals, *PyObject* *locals, int closeit, *PyCompilerFlags* *flags)
Return value: New reference. Similar to `PyRun_StringFlags()`, but the Python source code is read from *fp* instead of an in-memory string. *filename* should be the name of the file, it is decoded from the filesystem encoding (`sys.getfilesystemencoding()`). If *closeit* is true, the file is closed before `PyRun_FileExFlags()` returns.

*PyObject** **Py_CompileString**(const char *str, const char *filename, int start)
Return value: New reference. This is a simplified interface to `Py_CompileStringFlags()` below, leaving *flags* set to `NULL`.

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

Return value: New reference. This is a simplified interface to `Py_CompileStringExFlags()` below, with *optimize* set to `-1`.

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

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

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

버전 3.4에 추가.

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

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

버전 3.2에 추가.

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

Return value: New reference. This is a simplified interface to `PyEval_EvalCodeEx()`, with just the code object, and global and local variables. The other arguments are set to `NULL`.

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

Return value: New reference. Evaluate a precompiled code object, given a particular environment for its evaluation. This environment consists of a dictionary of global variables, a mapping object of local variables, arrays of arguments, keywords and defaults, a dictionary of default values for *keyword-only* arguments and a closure tuple of cells.

PyFrameObject

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

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

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

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

Return value: New reference. This is the main, unvarnished function of Python interpretation. It is literally 2000 lines long. The code object associated with the execution frame *f* is executed, interpreting bytecode and executing calls as needed. The additional *throwflag* parameter can mostly be ignored - if true, then it causes an exception to immediately be thrown; this is used for the `throw()` methods of generator objects.

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

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

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

int **Py_eval_input**

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

int **Py_file_input**

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

int **Py_single_input**

The start symbol from the Python grammar for a single statement; for use with `Py_CompileString()`.

This is the symbol used for the interactive interpreter loop.

struct **PyCompilerFlags**

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

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

```
struct PyCompilerFlags {
    int cf_flags;
}
```

int **CO_FUTURE_DIVISION**

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

CHAPTER 4

참조 횟수

이 섹션의 매크로는 파이썬 객체의 참조 횟수를 관리하는 데 사용됩니다.

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

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

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

> **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*)
Decrement the reference count for object *o*. The object may be NULL, in which case the macro has no effect; otherwise the effect is the same as for *Py_DECREF()*, and the same warning applies.

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

가비지 수집 중에 탐색 될 수 있는 변수의 값을 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.

CHAPTER 5

Exception Handling

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

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

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

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

5.1 Printing and clearing

void **PyErr_Clear**()
Clear the error indicator. If the error indicator is not set, there is no effect.

void **PyErr_PrintEx**(int *set_sys_last_vars*)
Print a standard traceback to `sys.stderr` and clear the error indicator. **Unless** the error is a `SystemExit`. In that case the no traceback is printed and Python process will exit with the error code specified by the `SystemExit` instance.

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

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

void **PyErr_Print**()
Alias for `PyErr_PrintEx(1)`.

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

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

An exception must be set when calling this function.

5.2 Raising exceptions

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

void **PyErr_SetString**(*PyObject* *type, const char *message)
This is the most common way to set the error indicator. The first argument specifies the exception type; it is normally one of the standard exceptions, e.g. `PyExc_RuntimeError`. You need not increment its reference count. The second argument is an error message; it is decoded from `'utf-8'`.

void **PyErr_SetObject**(*PyObject* *type, *PyObject* *value)
This function is similar to *PyErr_SetString()* but lets you specify an arbitrary Python object for the "value" of the exception.

*PyObject** **PyErr_Format**(*PyObject* *exception, const char *format, ...)
Return value: Always NULL. This function sets the error indicator and returns `NULL`. *exception* should be a Python exception class. The *format* and subsequent parameters help format the error message; they have the same meaning and values as in *PyUnicode_FromFormat()*. *format* is an ASCII-encoded string.

*PyObject** **PyErr_FormatV**(*PyObject* *exception, const char *format, va_list *vargs*)
Return value: Always NULL. Same as *PyErr_Format()*, but taking a `va_list` argument rather than a variable number of arguments.

버전 3.5에 추가.

void **PyErr_SetNone**(*PyObject* *type)
This is a shorthand for `PyErr_SetObject(type, Py_None)`.

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

*PyObject** **PyErr_NoMemory**()
Return value: Always NULL. This is a shorthand for `PyErr_SetNone(PyExc_MemoryError)`; it returns `NULL` so an object allocation function can write `return PyErr_NoMemory();` when it runs out of memory.

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

*PyObject** **PyErr_SetFromErrnoWithFilenameObject**(*PyObject* *type, *PyObject* *filenameObject)
Return value: Always NULL. Similar to *PyErr_SetFromErrno()*, with the additional behavior that if

filenameObject is not `NULL`, it is passed to the constructor of *type* as a third parameter. In the case of `OSError` exception, this is used to define the `filename` attribute of the exception instance.

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

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

버전 3.4에 추가.

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

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

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

Return value: Always NULL. This is a convenience function to raise `WindowsError`. If called with *ierr* of `0`, the error code returned by a call to `GetLastError()` is used instead. It calls the Win32 function `FormatMessage()` to retrieve the Windows description of error code given by *ierr* or `GetLastError()`, then it constructs a tuple object whose first item is the *ierr* value and whose second item is the corresponding error message (gotten from `FormatMessage()`), and then calls `PyErr_SetObject(PyExc_WindowsError, object)`. This function always returns `NULL`.

Availability: Windows.

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

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

Availability: Windows.

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

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

Availability: Windows.

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

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

Availability: Windows.

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

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

Availability: Windows.

버전 3.4에 추가.

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

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

Availability: Windows.

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

Return value: Always NULL. This is a convenience function to raise `ImportError`. *msg* will be set as the exception's message string. *name* and *path*, both of which can be `NULL`, will be set as the `ImportError`'s respective `name` and `path` attributes.

버전 3.3에 추가.

void **PyErr_SyntaxLocationObject**(*PyObject* *filename, int lineno, int col_offset)
Set file, line, and offset information for the current exception. If the current exception is not a SyntaxError, then it sets additional attributes, which make the exception printing subsystem think the exception is a SyntaxError.

버전 3.4에 추가.

void **PyErr_SyntaxLocationEx**(const char *filename, int lineno, int col_offset)
Like *PyErr_SyntaxLocationObject()*, but *filename* is a byte string decoded from the filesystem encoding (os.fsdecode()).

버전 3.2에 추가.

void **PyErr_SyntaxLocation**(const char *filename, int lineno)
Like *PyErr_SyntaxLocationEx()*, but the col_offset parameter is omitted.

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

5.3 Issuing warnings

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

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

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

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

*PyObject** **PyErr_SetImportErrorSubclass**(*PyObject* *exception, *PyObject* *msg, *PyObject* *name, *PyObject* *path)
Return value: Always NULL. Much like *PyErr_SetImportError()* but this function allows for specifying a subclass of ImportError to raise.

버전 3.6에 추가.

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

버전 3.4에 추가.

int **PyErr_WarnExplicit**(*PyObject* *category, const char *message, const char *filename, int lineno, const char *module, *PyObject* *registry)
Similar to *PyErr_WarnExplicitObject()* except that *message* and *module* are UTF-8 encoded strings,

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

int **PyErr_WarnFormat**(*PyObject* *category, Py_ssize_t stack_level, const char *format, ...)
Function similar to *PyErr_WarnEx()*, but use *PyUnicode_FromFormat()* to format the warning message. *format* is an ASCII-encoded string.

버전 3.2에 추가.

int **PyErr_ResourceWarning**(*PyObject* *source, Py_ssize_t stack_level, const char *format, ...)
Function similar to *PyErr_WarnFormat()*, but *category* is `ResourceWarning` and it passes *source* to `warnings.WarningMessage()`.

버전 3.6에 추가.

5.4 Querying the error indicator

*PyObject** **PyErr_Occurred**()
Return value: Borrowed reference. Test whether the error indicator is set. If set, return the exception *type* (the first argument to the last call to one of the `PyErr_Set*()` functions or to *PyErr_Restore()*). If not set, return `NULL`. You do not own a reference to the return value, so you do not need to *Py_DECREF()* it.

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

---

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

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

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

참고: This function is normally only used by code that needs to catch exceptions or by code that needs to save and restore the error indicator temporarily, e.g.:

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

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

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

---

void **PyErr_Restore**(*PyObject* *type, *PyObject* *value, *PyObject* *traceback)
Set the error indicator from the three objects. If the error indicator is already set, it is cleared first. If the objects are `NULL`, the error indicator is cleared. Do not pass a `NULL` type and non-`NULL` value or traceback. The exception type should be a class. Do not pass an invalid exception type or value. (Violating these rules will cause subtle problems later.) This call takes away a reference to each object: you must own a reference to

each object before the call and after the call you no longer own these references. (If you don't understand this, don't use this function. I warned you.)

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

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

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

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

void **PyErr_GetExcInfo**(*PyObject* **ptype, *PyObject* **pvalue, *PyObject* **ptraceback)
Retrieve the exception info, as known from `sys.exc_info()`. This refers to an exception that was *already caught*, not to an exception that was freshly raised. Returns new references for the three objects, any of which may be `NULL`. Does not modify the exception info state.

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

버전 3.3에 추가.

void **PyErr_SetExcInfo**(*PyObject* *type, *PyObject* *value, *PyObject* *traceback)
Set the exception info, as known from `sys.exc_info()`. This refers to an exception that was *already caught*, not to an exception that was freshly raised. This function steals the references of the arguments. To clear the exception state, pass `NULL` for all three arguments. For general rules about the three arguments, see *PyErr_Restore()*.

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

버전 3.3에 추가.

5.5 Signal Handling

int **PyErr_CheckSignals**()
This function interacts with Python's signal handling. It checks whether a signal has been sent to the processes and if so, invokes the corresponding signal handler. If the `signal` module is supported, this can invoke a signal handler written in Python. In all cases, the default effect for `SIGINT` is to raise the `KeyboardInterrupt` exception. If an exception is raised the error indicator is set and the function returns `-1`; otherwise the function returns `0`. The error indicator may or may not be cleared if it was previously set.

void **PyErr_SetInterrupt**()
Simulate the effect of a `SIGINT` signal arriving. The next time *PyErr_CheckSignals()* is called, the

Python signal handler for `SIGINT` will be called.

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

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

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

버전 3.5에서 변경: On Windows, the function now also supports socket handles.

5.6 Exception Classes

*PyObject** **PyErr_NewException**(const char **name*, *PyObject* **base*, *PyObject* **dict*)
Return value: New reference. This utility function creates and returns a new exception class. The *name* argument must be the name of the new exception, a C string of the form `module.classname`. The *base* and *dict* arguments are normally `NULL`. This creates a class object derived from `Exception` (accessible in C as `PyExc_Exception`).

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

*PyObject** **PyErr_NewExceptionWithDoc**(const char **name*, const char **doc*, *PyObject* **base*, *PyObject* **dict*)
Return value: New reference. Same as *PyErr_NewException()*, except that the new exception class can easily be given a docstring: If *doc* is non-`NULL`, it will be used as the docstring for the exception class.

버전 3.2에 추가.

5.7 Exception Objects

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

int **PyException_SetTraceback**(*PyObject* **ex*, *PyObject* **tb*)
Set the traceback associated with the exception to *tb*. Use `Py_None` to clear it.

*PyObject** **PyException_GetContext**(*PyObject* **ex*)
Return value: New reference. Return the context (another exception instance during whose handling *ex* was raised) associated with the exception as a new reference, as accessible from Python through `__context__`. If there is no context associated, this returns `NULL`.

void **PyException_SetContext**(*PyObject* **ex*, *PyObject* **ctx*)
Set the context associated with the exception to *ctx*. Use `NULL` to clear it. There is no type check to make sure that *ctx* is an exception instance. This steals a reference to *ctx*.

*PyObject** **PyException_GetCause**(*PyObject* **ex*)
Return value: New reference. Return the cause (either an exception instance, or `None`, set by `raise ... from ...`) associated with the exception as a new reference, as accessible from Python through `__cause__`.

void **PyException_SetCause**(*PyObject* **ex*, *PyObject* **cause*)
Set the cause associated with the exception to *cause*. Use `NULL` to clear it. There is no type check to make sure that *cause* is either an exception instance or `None`. This steals a reference to *cause*.

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

5.8 Unicode Exception Objects

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

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

*PyObject** **PyUnicodeEncodeError_Create**(const char *encoding, const *Py_UNICODE* *object, Py_ssize_t *length*, Py_ssize_t *start*, Py_ssize_t *end*, const char *reason*)
Return value: New reference. Create a `UnicodeEncodeError` object with the attributes *encoding*, *object*, *length*, *start*, *end* and *reason*. *encoding* and *reason* are UTF-8 encoded strings.

*PyObject** **PyUnicodeTranslateError_Create**(const *Py_UNICODE* *object, Py_ssize_t *length*, Py_ssize_t *start*, Py_ssize_t *end*, const char *reason*)
Return value: New reference. Create a `UnicodeTranslateError` object with the attributes *object*, *length*, *start*, *end* and *reason*. *reason* is a UTF-8 encoded string.

*PyObject** **PyUnicodeDecodeError_GetEncoding**(*PyObject* *exc*)
*PyObject** **PyUnicodeEncodeError_GetEncoding**(*PyObject* *exc*)
Return value: New reference. Return the *encoding* attribute of the given exception object.

*PyObject** **PyUnicodeDecodeError_GetObject**(*PyObject* *exc*)
*PyObject** **PyUnicodeEncodeError_GetObject**(*PyObject* *exc*)
*PyObject** **PyUnicodeTranslateError_GetObject**(*PyObject* *exc*)
Return value: New reference. Return the *object* attribute of the given exception object.

int **PyUnicodeDecodeError_GetStart**(*PyObject* *exc*, Py_ssize_t **start*)
int **PyUnicodeEncodeError_GetStart**(*PyObject* *exc*, Py_ssize_t **start*)
int **PyUnicodeTranslateError_GetStart**(*PyObject* *exc*, Py_ssize_t **start*)
Get the *start* attribute of the given exception object and place it into **start*. *start* must not be `NULL`. Return `0` on success, `-1` on failure.

int **PyUnicodeDecodeError_SetStart**(*PyObject* *exc*, Py_ssize_t *start*)
int **PyUnicodeEncodeError_SetStart**(*PyObject* *exc*, Py_ssize_t *start*)
int **PyUnicodeTranslateError_SetStart**(*PyObject* *exc*, Py_ssize_t *start*)
Set the *start* attribute of the given exception object to *start*. Return `0` on success, `-1` on failure.

int **PyUnicodeDecodeError_GetEnd**(*PyObject* *exc*, Py_ssize_t **end*)
int **PyUnicodeEncodeError_GetEnd**(*PyObject* *exc*, Py_ssize_t **end*)
int **PyUnicodeTranslateError_GetEnd**(*PyObject* *exc*, Py_ssize_t **end*)
Get the *end* attribute of the given exception object and place it into **end*. *end* must not be `NULL`. Return `0` on success, `-1` on failure.

int **PyUnicodeDecodeError_SetEnd**(*PyObject* *exc*, Py_ssize_t *end*)
int **PyUnicodeEncodeError_SetEnd**(*PyObject* *exc*, Py_ssize_t *end*)
int **PyUnicodeTranslateError_SetEnd**(*PyObject* *exc*, Py_ssize_t *end*)
Set the *end* attribute of the given exception object to *end*. Return `0` on success, `-1` on failure.

*PyObject** **PyUnicodeDecodeError_GetReason**(*PyObject* *exc*)
*PyObject** **PyUnicodeEncodeError_GetReason**(*PyObject* *exc*)
*PyObject** **PyUnicodeTranslateError_GetReason**(*PyObject* *exc*)
Return value: New reference. Return the *reason* attribute of the given exception object.

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

int **PyUnicodeTranslateError_SetReason**(*PyObject* *exc, const char *reason)
Set the *reason* attribute of the given exception object to *reason*. Return `0` on success, `-1` on failure.

5.9 Recursion Control

These two functions provide a way to perform safe recursive calls at the C level, both in the core and in extension modules. They are needed if the recursive code does not necessarily invoke Python code (which tracks its recursion depth automatically).

int **Py_EnterRecursiveCall**(const char *where)
Marks a point where a recursive C-level call is about to be performed.

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

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

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

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

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

int **Py_ReprEnter**(*PyObject* *object)
Called at the beginning of the *tp_repr* implementation to detect cycles.

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

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

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

void **Py_ReprLeave**(*PyObject* *object)
Ends a *Py_ReprEnter()*. Must be called once for each invocation of *Py_ReprEnter()* that returns zero.

5.10 Standard Exceptions

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

C Name	Python Name	Notes
`PyExc_BaseException`	`BaseException`	(1)
`PyExc_Exception`	`Exception`	(1)
`PyExc_ArithmeticError`	`ArithmeticError`	(1)
`PyExc_AssertionError`	`AssertionError`	
`PyExc_AttributeError`	`AttributeError`	
`PyExc_BlockingIOError`	`BlockingIOError`	

다음 페이지에 계속

표 1 – 이전 페이지에서 계속

C Name	Python Name	Notes
PyExc_BrokenPipeError	BrokenPipeError	
PyExc_BufferError	BufferError	
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 and PyExc_TimeoutError were introduced following **PEP 3151**.

버전 3.5에 추가: `PyExc_StopAsyncIteration` and `PyExc_RecursionError`.

버전 3.6에 추가: `PyExc_ModuleNotFoundError`.

These are compatibility aliases to `PyExc_OSError`:

C Name	Notes
`PyExc_EnvironmentError`	
`PyExc_IOError`	
`PyExc_WindowsError`	(3)

버전 3.3에서 변경: These aliases used to be separate exception types.

Notes:

(1) This is a base class for other standard exceptions.

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

5.11 Standard Warning Categories

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

C Name	Python Name	Notes
`PyExc_Warning`	`Warning`	(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`.

Notes:

(1) This is a base class for other standard warning categories.

CHAPTER 6

유틸리티

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 Operating System Utilities

*PyObject** **PyOS_FSPath**(*PyObject *path*)
Return value: New reference. Return the file system representation for *path*. If the object is a `str` or `bytes` object, then its reference count is incremented. If the object implements the `os.PathLike` interface, then `__fspath__()` is returned as long as it is a `str` or `bytes` object. Otherwise `TypeError` is raised and `NULL` is returned.

버전 3.6에 추가.

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

void **PyOS_BeforeFork**()
Function to prepare some internal state before a process fork. This should be called before calling `fork()` or any similar function that clones the current process. Only available on systems where `fork()` is defined.

버전 3.7에 추가.

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

버전 3.7에 추가.

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

버전 3.7에 추가.

더 보기:

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

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

버전 3.7부터 폐지: This function is superseded by *PyOS_AfterFork_Child()*.

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

PyOS_sighandler_t **PyOS_getsig**(int *i*)
Return the current signal handler for signal *i*. This is a thin wrapper around either `sigaction()` or `signal()`. Do not call those functions directly! `PyOS_sighandler_t` is a typedef alias for `void (*)(int)`.

PyOS_sighandler_t **PyOS_setsig**(int *i*, PyOS_sighandler_t *h*)
Set the signal handler for signal *i* to be *h*; return the old signal handler. This is a thin wrapper around either `sigaction()` or `signal()`. Do not call those functions directly! `PyOS_sighandler_t` is a typedef alias for `void (*)(int)`.

wchar_t* **Py_DecodeLocale**(const char* *arg*, size_t **size*)
Decode a byte string from the locale encoding with the surrogateescape error handler: undecodable bytes are decoded as characters in range U+DC80..U+DCFF. If a byte sequence can be decoded as a surrogate character, escape the bytes using the surrogateescape error handler instead of decoding them.

Encoding, highest priority to lowest priority:

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

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

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

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

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

더 보기:

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

버전 3.5에 추가.

버전 3.7에서 변경: The function now uses the UTF-8 encoding in the UTF-8 mode.

char* **Py_EncodeLocale**(const wchar_t **text*, size_t **error_pos*)
Encode a wide character string to the locale encoding with the surrogateescape error handler: surrogate characters in the range U+DC80..U+DCFF are converted to bytes 0x80..0xFF.

Encoding, highest priority to lowest priority:

- `UTF-8` on macOS and Android;

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

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

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

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

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

버전 3.7에서 변경: The function now uses the UTF-8 encoding in the UTF-8 mode.

더 보기:

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

버전 3.5에 추가.

버전 3.7에서 변경: The function now supports the UTF-8 mode.

6.2 System Functions

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

PyObject ***PySys_GetObject**(const char **name*)
Return value: Borrowed reference. Return the object *name* from the `sys` module or `NULL` if it does not exist, without setting an exception.

int **PySys_SetObject**(const char **name*, *PyObject* **v*)
Set *name* in the `sys` module to *v* unless *v* is `NULL`, in which case *name* is deleted from the sys module. Returns `0` on success, `-1` on error.

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

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

void **PySys_AddWarnOptionUnicode**(*PyObject* **unicode*)
Append *unicode* to `sys.warnoptions`.

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

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

void **PySys_WriteStdout**(const char **format*, ...)
Write the output string described by *format* to `sys.stdout`. No exceptions are raised, even if truncation occurs (see below).

format should limit the total size of the formatted output string to 1000 bytes or less – after 1000 bytes, the output string is truncated. In particular, this means that no unrestricted "%s" formats should occur; these should be limited using "%.<N>s" where <N> is a decimal number calculated so that <N> plus the maximum size of

other formatted text does not exceed 1000 bytes. Also watch out for "%f", which can print hundreds of digits for very large numbers.

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

void **PySys_WriteStderr**(const char *format, ...)
As *PySys_WriteStdout()*, but write to `sys.stderr` or *stderr* instead.

void **PySys_FormatStdout**(const char *format, ...)
Function similar to PySys_WriteStdout() but format the message using *PyUnicode_FromFormatV()* and don't truncate the message to an arbitrary length.

버전 3.2에 추가.

void **PySys_FormatStderr**(const char *format, ...)
As *PySys_FormatStdout()*, but write to `sys.stderr` or *stderr* instead.

버전 3.2에 추가.

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

버전 3.2에 추가.

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

버전 3.2에 추가.

6.3 Process Control

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

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

버전 3.6에서 변경: Errors from finalization no longer ignored.

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

6.4 모듈 임포트 하기

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

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.

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

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.

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

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, name은 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 훅 함수"를 호출하는 고수준 인터페이스입니다 (명시적인 *level* 0을 사용하는데, 절대 임포트를 뜻합니다). 현재 전역의 `__builtins__`에 있는 `__import__()` 함수를 호출합니다. 이는 현재 환경에 설치된 임포트 훅을 사용하여 임포트가 수행됨을 의미합니다.

이 함수는 항상 절대 임포트를 사용합니다.

*PyObject** **PyImport_ReloadModule**(*PyObject* *m)
Return value: New reference. Reload a module. Return a new reference to the reloaded module, or `NULL` with an exception set on failure (the module still exists in this case).

*PyObject** **PyImport_AddModuleObject**(*PyObject* *name)
Return value: Borrowed reference. Return the module object corresponding to a module name. The *name* argument may be of the form `package.module`. First check the modules dictionary if there's one there,

and if not, create a new one and insert it in the modules dictionary. Return NULL with an exception set on failure.

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, name은 유니코드 객체 대신 UTF-8로 인코딩된 문자열입니다.

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

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. 에러 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_tag의 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. Return the already imported module with the given name. If the module has not been imported yet then returns NULL but does not set an error. Returns NULL and sets an error if the lookup failed.

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 캐시 되지 않았으면, 경로 항목을 처리할 수 있는 훅이 발견될 때까지 sys.path_hooks를 탐색합니다. 훅이 없으면 None을 반환합니다; 이것은 호출자에게 경로 기반 파인더가 이 경로 항목에 대한 파인더를 찾을 수 없음을 알려줍니다. sys.path_importer_cache에 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_ImportFrozenModuleObject**(*PyObject *name*)
Return value: New reference. *name*이라는 이름의 프로즌 모듈(frozen module)을 로드합니다. 성공하면 1을, 모듈을 찾지 못하면 0을, 초기화에 실패하면 예외를 설정하고 -1을 반환합니다. 로드가 성공할 때 임포트 된 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**
This pointer is initialized to point to an array of `struct _frozen` records, terminated by one whose members are all `NULL` or zero. When a frozen module is imported, it is searched in this table. Third-party code could play tricks with this to provide a dynamically created collection of frozen modules.

int **PyImport_AppendInittab**(const char **name*, *PyObject** (**initfunc*)(void))
기존의 내장 모듈 테이블에 단일 모듈을 추가합니다. 이것은 *PyImport_ExtendInittab()*을 감싸는 편리한 래퍼인데, 테이블을 확장할 수 없으면 `-1`을 반환합니다. 새 모듈은 *name*이라는 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 제공 할 수 있습니다. 구조체는 `Include/import.h`에서 다음과 같이 정의됩니다:

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

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

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.

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

void **PyMarshal_WriteLongToFile**(long *value*, FILE **file*, int *version*)
`long` 정수 *value*를 *file*로 마샬합니다. *value*의 최하위 32비트 만 기록합니다; 기본 `long` 형의 크기와 관계없이. *version*은 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 *value*를 *file*로 마샬합니다. *version*은 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. *version*은 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 `long`을 반환합니다. 이 함수를 사용하면 `long`의 기본 크기와 관계없이 32비트 값만 읽을 수 있습니다.

에러 시, 적절한 예외(`EOFError`)를 설정하고 `-1`을 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.

에러 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.

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

*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.

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

*PyObject** **PyMarshal_ReadObjectFromString**(const char *data, Py_ssize_t *len*)
Return value: New reference. *data*가 가리키는 *len* 바이트를 포함하는 바이트 버퍼의 데이터 스트림에서 파이썬 객체를 반환합니다.

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

6.6 Parsing arguments and building values

These functions are useful when creating your own extensions functions and methods. Additional information and examples are available in extending-index.

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

6.6.1 Parsing arguments

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

Strings and buffers

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

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

However, when a *Py_buffer* structure gets filled, the underlying buffer is locked so that the caller can subsequently use the buffer even inside a *Py_BEGIN_ALLOW_THREADS* block without the risk of mutable data being resized or destroyed. As a result, **you have to call** *PyBuffer_Release()* after you have finished processing the data (or in any early abort case).

Unless otherwise stated, buffers are not NUL-terminated.

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

참고: For all # variants of formats (s#, y#, etc.), the type of the length argument (int or Py_ssize_t) is controlled by defining the macro PY_SSIZE_T_CLEAN before including Python.h. If the macro was defined, length is a Py_ssize_t rather than an int. This behavior will change in a future Python version to only support Py_ssize_t and drop int support. It is best to always define PY_SSIZE_T_CLEAN.

---

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

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

버전 3.5에서 변경: Previously, TypeError was raised when embedded null code points were encountered in the Python string.

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

s# (str, read-only *bytes-like object*) [const char *, int or Py_ssize_t] Like s*, except that it doesn't accept mutable objects. The result is stored into two C variables, the first one a pointer to a C string, the second one its length. The string may contain embedded null bytes. Unicode objects are converted to C strings using 'utf-8' encoding.

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

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

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

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

버전 3.5에서 변경: Previously, TypeError was raised when embedded null bytes were encountered in the bytes buffer.

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

y# (read-only *bytes-like object*) [const char *, int or Py_ssize_t] This variant on s# doesn't accept Unicode objects, only bytes-like objects.

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

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

u (str) [const Py_UNICODE *] Convert a Python Unicode object to a C pointer to a NUL-terminated buffer of Unicode characters. You must pass the address of a *Py_UNICODE* pointer variable, which will be filled with

the pointer to an existing Unicode buffer. Please note that the width of a *Py_UNICODE* character depends on compilation options (it is either 16 or 32 bits). The Python string must not contain embedded null code points; if it does, a `ValueError` exception is raised.

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

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

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

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

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

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

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

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

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

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

es (str) [const char *encoding, char **buffer] This variant on `s` is used for encoding Unicode into a character buffer. It only works for encoded data without embedded NUL bytes.

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

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

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

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

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

There are two modes of operation:

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

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

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

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

Numbers

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

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

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

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

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

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

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

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

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

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

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

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

버전 3.3에서 변경: Allow `bytearray` objects.

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

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

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

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

Other objects

O (object) [PyObject *] Store a Python object (without any conversion) in a C object pointer. The C program thus receives the actual object that was passed. The object's reference count is not increased. The pointer stored is not NULL.

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

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

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

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

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

버전 3.1에서 변경: Py_CLEANUP_SUPPORTED was added.

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

버전 3.3에 추가.

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

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

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

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

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

버전 3.3에 추가.

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

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

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

Additional arguments passed to these functions must be addresses of variables whose type is determined by the format string; these are used to store values from the input tuple. There are a few cases, as described in the list of format units

above, where these parameters are used as input values; they should match what is specified for the corresponding format unit in that case.

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

API Functions

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

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

int **PyArg_ParseTupleAndKeywords**(*PyObject* **args*, *PyObject* **kw*, const char **format*, char **keywords[]*, ...)
Parse the parameters of a function that takes both positional and keyword parameters into local variables. The *keywords* argument is a `NULL`-terminated array of keyword parameter names. Empty names denote *positional-only parameters*. Returns true on success; on failure, it returns false and raises the appropriate exception.

버전 3.6에서 변경: Added support for *positional-only parameters*.

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

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

버전 3.2에 추가.

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

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

This is an example of the use of this function, taken from the sources for the `_weakref` helper module for weak references:

```
static PyObject *
weakref_ref(PyObject *self, PyObject *args)
{
    PyObject *object;
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    PyObject *callback = NULL;
    PyObject *result = NULL;

    if (PyArg_UnpackTuple(args, "ref", 1, 2, &object, &callback)) {
        result = PyWeakref_NewRef(object, callback);
    }
    return result;
}
```

The call to *PyArg_UnpackTuple()* in this example is entirely equivalent to this call to *PyArg_ParseTuple()*:

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

6.6.2 Building values

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

Return value: New reference. Create a new value based on a format string similar to those accepted by the *PyArg_Parse*()* family of functions and a sequence of values. Returns the value or NULL in the case of an error; an exception will be raised if NULL is returned.

Py_BuildValue() does not always build a tuple. It builds a tuple only if its format string contains two or more format units. If the format string is empty, it returns None; if it contains exactly one format unit, it returns whatever object is described by that format unit. To force it to return a tuple of size 0 or one, parenthesize the format string.

When memory buffers are passed as parameters to supply data to build objects, as for the s and s# formats, the required data is copied. Buffers provided by the caller are never referenced by the objects created by *Py_BuildValue()*. In other words, if your code invokes malloc() and passes the allocated memory to *Py_BuildValue()*, your code is responsible for calling free() for that memory once *Py_BuildValue()* returns.

In the following description, the quoted form is the format unit; the entry in (round) parentheses is the Python object type that the format unit will return; and the entry in [square] brackets is the type of the C value(s) to be passed.

The characters space, tab, colon and comma are ignored in format strings (but not within format units such as s#). This can be used to make long format strings a tad more readable.

s (str or None) [const char *] Convert a null-terminated C string to a Python str object using 'utf-8' encoding. If the C string pointer is NULL, None is used.

s# (str or None) [const char *, int or Py_ssize_t] Convert a C string and its length to a Python str object using 'utf-8' encoding. If the C string pointer is NULL, the length is ignored and None is returned.

y (bytes) [const char *] This converts a C string to a Python bytes object. If the C string pointer is NULL, None is returned.

y# (bytes) [const char *, int or Py_ssize_t] This converts a C string and its lengths to a Python object. If the C string pointer is NULL, None is returned.

z (str or None) [const char *] Same as s.

z# (str or None) [const char *, int or Py_ssize_t] Same as s#.

u (str) [const wchar_t *] Convert a null-terminated wchar_t buffer of Unicode (UTF-16 or UCS-4) data to a Python Unicode object. If the Unicode buffer pointer is NULL, None is returned.

u# (str) [const wchar_t *, int or Py_ssize_t] Convert a Unicode (UTF-16 or UCS-4) data buffer and its length to a Python Unicode object. If the Unicode buffer pointer is NULL, the length is ignored and None is returned.

U (str or None) [const char *] Same as `s`.

U# (str or None) [const char *, int or Py_ssize_t] Same as `s#`.

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

b (int) [char] Convert a plain C `char` to a Python integer object.

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

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

B (int) [unsigned char] Convert a C `unsigned char` to a Python integer object.

H (int) [unsigned short int] Convert a C `unsigned short int` to a Python integer object.

I (int) [unsigned int] Convert a C `unsigned int` to a Python integer object.

k (int) [unsigned long] Convert a C `unsigned long` to a Python integer object.

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

K (int) [unsigned long long] Convert a C `unsigned long long` to a Python integer object.

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

c (bytes of length 1) [char] Convert a C `int` representing a byte to a Python `bytes` object of length 1.

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

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

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

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

O (object) [PyObject *] Pass a Python object untouched (except for its reference count, which is incremented by one). If the object passed in is a `NULL` pointer, it is assumed that this was caused because the call producing the argument found an error and set an exception. Therefore, *`Py_BuildValue()`* will return `NULL` but won't raise an exception. If no exception has been raised yet, `SystemError` is set.

S (object) [PyObject *] Same as `O`.

N (object) [PyObject *] Same as `O`, except it doesn't increment the reference count on the object. Useful when the object is created by a call to an object constructor in the argument list.

O& (object) [*converter*, *anything*] Convert *anything* to a Python object through a *converter* function. The function is called with *anything* (which should be compatible with `void *`) as its argument and should return a "new" Python object, or `NULL` if an error occurred.

(items) (tuple) [*matching-items*] Convert a sequence of C values to a Python tuple with the same number of items.

[items] (list) [*matching-items*] Convert a sequence of C values to a Python list with the same number of items.

{items} (dict) [*matching-items*] Convert a sequence of C values to a Python dictionary. Each pair of consecutive C values adds one item to the dictionary, serving as key and value, respectively.

If there is an error in the format string, the `SystemError` exception is set and `NULL` returned.

*PyObject** **Py_VaBuildValue**(const char **format*, va_list *vargs*)
Return value: New reference. Identical to *`Py_BuildValue()`*, except that it accepts a va_list rather than a variable number of arguments.

6.7 문자열 변환과 포매팅

숫자 변환과 포맷된 문자열 출력을 위한 함수.

int **PyOS_snprintf**(char *str, size_t size, const char *format, ...)
포맷 문자열 *format* 과 추가 인자에 따라 *size* 바이트를 넘지 않도록 *str*로 출력합니다. 유닉스 매뉴얼 페이지 *snprintf(2)*를 보십시오.

int **PyOS_vsnprintf**(char *str, size_t size, const char *format, va_list va)
포맷 문자열 *format* 과 가변 인자 목록 *va*에 따라 *size* 바이트를 넘지 않도록 *str*로 출력합니다. 유닉스 매뉴얼 페이지 *vsnprintf(2)*를 보십시오.

*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.

래퍼는 반환 시 *str*[*size*-1]이 항상 `'\0'`이 되도록 합니다. str에 *size* 바이트(후행 `'\0'` 포함)를 초과해서 쓰지 않습니다. 두 함수 모두 `str != NULL`, `size > 0` 및 `format != NULL`을 요구합니다.

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

이 함수들의 반환 값(*rv*)은 다음과 같이 해석되어야 합니다:

- `0 <= rv < size` 일 때, 출력 변환에 성공했으며 *rv* 문자가 *str*에 기록되었습니다 (*str*[*rv*]의 후행 `'\0'` 바이트 제외).
- `rv >= size` 일 때, 출력 변환이 잘렸고 성공하려면 `rv + 1` 바이트의 버퍼가 필요합니다. *str*[*size*-1]은 이때 `'\0'`입니다.
- `rv < 0` 일 때, "뭔가 나쁜 일이 일어났습니다." 이때도 *str*[*size*-1]은 `'\0'`이지만, *str*의 나머지는 정의되지 않습니다. 에러의 정확한 원인은 하부 플랫폼에 따라 다릅니다.

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.

endptr이 `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, ValueError를 발생시키고 `-1.0`을 반환합니다.

`s`가 float에 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`를 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 can be zero or more of the values `Py_DTSF_SIGN`, `Py_DTSF_ADD_DOT_0`, or `Py_DTSF_ALT`, or-ed together:

- `Py_DTSF_SIGN` means to always precede the returned string with a sign character, even if *val* is non-negative.
- `Py_DTSF_ADD_DOT_0` means to ensure that the returned string will not look like an integer.
- `Py_DTSF_ALT` means to apply "alternate" formatting rules. See the documentation for the *PyOS_snprintf()* `'#'` specifier for details.

If *ptype* is non-`NULL`, then the value it points to will be set to one of `Py_DTST_FINITE`, `Py_DTST_INFINITE`, or `Py_DTST_NAN`, signifying that *val* is a finite number, an infinite number, or not a number, respectively.

The return value is a pointer to *buffer* with the converted string or `NULL` if the conversion failed. The caller is responsible for freeing the returned string by calling *PyMem_Free()*.

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_strnicmp**(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. Return a dictionary of the local variables in the current execution frame, or `NULL` if no frame is currently executing.

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

*PyFrameObject** **PyEval_GetFrame**()
Return value: Borrowed reference. Return the current thread state's frame, which is `NULL` if no frame is currently executing.

int **PyFrame_GetLineNumber**(*PyFrameObject *frame*)
*frame*이 현재 실행 중인 줄 번호를 반환합니다.

const char* **PyEval_GetFuncName**(*PyObject *func*)
*func*가 함수, 클래스 또는 인스턴스 객체면 *func*의 이름을 반환하고, 그렇지 않으면 *func*의 형의 이름을 반환합니다.

const char* **PyEval_GetFuncDesc**(*PyObject *func*)
*func*의 형에 따라 설명 문자열을 반환합니다. 반환 값에는 함수 및 메서드의 "()", " constructor", " instance" 및 " object" 가 포함됩니다. *PyEval_GetFuncName()*의 결과와 이어붙이면 *func*의 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 *encoding*에 대해 등록된 코덱이 있는지에 따라 `1` 이나 `0`을 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.

object is passed through the encoder function found for the given *encoding* using the error handling method defined by *errors*. *errors* may be `NULL` to use the default method defined for the codec. Raises a `LookupError` if no encoder can be found.

*PyObject** **PyCodec_Decode**(*PyObject *object*, const char *encoding*, const char *errors*)
Return value: New reference. 일반 코덱 기반 디코딩 API.

object is passed through the decoder function found for the given *encoding* using the error handling method defined by *errors*. *errors* may be `NULL` to use the default method defined for the codec. Raises a `LookupError` if no encoder can be found.

6.9.1 코덱 조회 API

In the following functions, the *encoding* string is looked up converted to all lower-case characters, which makes encodings looked up through this mechanism effectively case-insensitive. If no codec is found, a `KeyError` is set and `NULL` returned.

*PyObject** **PyCodec_Encoder**(const char *encoding*)
Return value: New reference. 주어진 *encoding*에 대한 인코더 함수를 가져옵니다.

*PyObject** **PyCodec_Decoder**(const char *encoding*)
Return value: New reference. 주어진 *encoding*에 대한 디코더 함수를 가져옵니다.

*PyObject** **PyCodec_IncrementalEncoder**(const char *encoding*, const char *errors*)
Return value: New reference. 지정된 *encoding*에 대한 `IncrementalEncoder` 객체를 가져옵니다.

*PyObject** **PyCodec_IncrementalDecoder**(const char *encoding*, const char *errors*)
Return value: New reference. 지정된 *encoding*에 대한 `IncrementalDecoder` 객체를 가져옵니다.

*PyObject** **PyCodec_StreamReader**(const char *encoding*, *PyObject *stream*, const char *errors*)
Return value: New reference. 지정된 *encoding*에 대한 `StreamReader` 팩토리 함수를 가져옵니다.

*PyObject** **PyCodec_StreamWriter**(const char *encoding*, *PyObject *stream*, const char *errors*)
Return value: New reference. 지정된 *encoding*에 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 *error*를 등록합니다. 코덱이 인코딩할 수 없는 문자/디코딩할 수 없는 바이트열을 발견하고, 인코드/디코드 함수를 호출할 때 *name*이 error 매개 변수로 지정되었을 때 이 콜백 함수를 호출합니다.

콜백은 하나의 인자로 `UnicodeEncodeError`, `UnicodeDecodeError` 또는 `UnicodeTranslateError`의 인스턴스를 받아들이는데, 문제가 되는 문자나 바이트의 시퀀스와 이들의 원본 문자열에서의 오프셋에 대한 정보를 담고 있습니다 (이 정보를 추출하는 함수는 *Unicode Exception Objects*를 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 `0`을, 에러면 `-1`을 반환합니다.

*PyObject** **PyCodec_LookupError**(const char *name)
Return value: New reference. Lookup the error handling callback function registered under *name*. As a special case `NULL` can be passed, in which case the error handling callback for "strict" will be returned.

*PyObject** **PyCodec_StrictErrors**(*PyObject* *exc)
Return value: Always NULL. *exc*를 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+FFFD`로 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.

CHAPTER 7

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 Object Protocol

*PyObject** **Py_NotImplemented**
The NotImplemented singleton, used to signal that an operation is not implemented for the given type combination.

Py_RETURN_NOTIMPLEMENTED
Properly handle returning *Py_NotImplemented* from within a C function (that is, increment the reference count of NotImplemented and return it).

int **PyObject_Print**(*PyObject* *o, FILE *fp, int *flags*)
Print an object *o*, on file *fp*. Returns -1 on error. The flags argument is used to enable certain printing options. The only option currently supported is Py_PRINT_RAW; if given, the str() of the object is written instead of the repr().

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

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

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

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

*PyObject** **PyObject_GetAttr**(*PyObject* *o, *PyObject* *attr_name*)
Return value: New reference. Retrieve an attribute named *attr_name* from object *o*. Returns the attribute value on success, or NULL on failure. This is the equivalent of the Python expression o.attr_name.

*PyObject** **PyObject_GetAttrString**(*PyObject* *o, const char *attr_name)
Return value: New reference. Retrieve an attribute named *attr_name* from object *o*. Returns the attribute value on success, or `NULL` on failure. This is the equivalent of the Python expression `o.attr_name`.

*PyObject** **PyObject_GenericGetAttr**(*PyObject* *o, *PyObject* *name)
Return value: New reference. Generic attribute getter function that is meant to be put into a type object's `tp_getattro` slot. It looks for a descriptor in the dictionary of classes in the object's MRO as well as an attribute in the object's `__dict__` (if present). As outlined in descriptors, data descriptors take preference over instance attributes, while non-data descriptors don't. Otherwise, an `AttributeError` is raised.

int **PyObject_SetAttr**(*PyObject* *o, *PyObject* *attr_name, *PyObject* *v)
Set the value of the attribute named *attr_name*, for object *o*, to the value *v*. Raise an exception and return `-1` on failure; return `0` on success. This is the equivalent of the Python statement `o.attr_name = v`.

If *v* is `NULL`, the attribute is deleted, however this feature is deprecated in favour of using *PyObject_DelAttr()*.

int **PyObject_SetAttrString**(*PyObject* *o, const char *attr_name, *PyObject* *v)
Set the value of the attribute named *attr_name*, for object *o*, to the value *v*. Raise an exception and return `-1` on failure; return `0` on success. This is the equivalent of the Python statement `o.attr_name = v`.

If *v* is `NULL`, the attribute is deleted, however this feature is deprecated in favour of using *PyObject_DelAttrString()*.

int **PyObject_GenericSetAttr**(*PyObject* *o, *PyObject* *name, *PyObject* *value)
Generic attribute setter and deleter function that is meant to be put into a type object's *tp_setattro* slot. It looks for a data descriptor in the dictionary of classes in the object's MRO, and if found it takes preference over setting or deleting the attribute in the instance dictionary. Otherwise, the attribute is set or deleted in the object's `__dict__` (if present). On success, `0` is returned, otherwise an `AttributeError` is raised and `-1` is returned.

int **PyObject_DelAttr**(*PyObject* *o, *PyObject* *attr_name)
Delete attribute named *attr_name*, for object *o*. Returns `-1` on failure. This is the equivalent of the Python statement `del o.attr_name`.

int **PyObject_DelAttrString**(*PyObject* *o, const char *attr_name)
Delete attribute named *attr_name*, for object *o*. Returns `-1` on failure. This is the equivalent of the Python statement `del o.attr_name`.

*PyObject** **PyObject_GenericGetDict**(*PyObject* *o, void *context)
Return value: New reference. A generic implementation for the getter of a `__dict__` descriptor. It creates the dictionary if necessary.

버전 3.3에 추가.

int **PyObject_GenericSetDict**(*PyObject* *o, *PyObject* *value, void *context)
A generic implementation for the setter of a `__dict__` descriptor. This implementation does not allow the dictionary to be deleted.

버전 3.3에 추가.

*PyObject** **PyObject_RichCompare**(*PyObject* *o1, *PyObject* *o2, int *opid*)
Return value: New reference. Compare the values of *o1* and *o2* using the operation specified by *opid*, which must be one of `Py_LT`, `Py_LE`, `Py_EQ`, `Py_NE`, `Py_GT`, or `Py_GE`, corresponding to `<`, `<=`, `==`, `!=`, `>`, or `>=` respectively. This is the equivalent of the Python expression `o1 op o2`, where `op` is the operator corresponding to *opid*. Returns the value of the comparison on success, or `NULL` on failure.

int **PyObject_RichCompareBool**(*PyObject* *o1, *PyObject* *o2, int *opid*)
Compare the values of *o1* and *o2* using the operation specified by *opid*, which must be one of `Py_LT`, `Py_LE`, `Py_EQ`, `Py_NE`, `Py_GT`, or `Py_GE`, corresponding to `<`, `<=`, `==`, `!=`, `>`, or `>=` respectively. Returns `-1` on error, `0` if the result is false, `1` otherwise. This is the equivalent of the Python expression `o1 op o2`, where `op` is the operator corresponding to *opid*.

참고: If *o1* and *o2* are the same object, *PyObject_RichCompareBool()* will always return 1 for Py_EQ and 0 for Py_NE.

*PyObject** **PyObject_Repr** (*PyObject *o*)
Return value: New reference. Compute a string representation of object *o*. Returns the string representation on success, NULL on failure. This is the equivalent of the Python expression repr(o). Called by the repr() built-in function.

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

*PyObject** **PyObject_ASCII** (*PyObject *o*)
Return value: New reference. As *PyObject_Repr()*, compute a string representation of object *o*, but escape the non-ASCII characters in the string returned by *PyObject_Repr()* with \x, \u or \U escapes. This generates a string similar to that returned by *PyObject_Repr()* in Python 2. Called by the ascii() built-in function.

*PyObject** **PyObject_Str** (*PyObject *o*)
Return value: New reference. Compute a string representation of object *o*. Returns the string representation on success, NULL on failure. This is the equivalent of the Python expression str(o). Called by the str() built-in function and, therefore, by the print() function.

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

*PyObject** **PyObject_Bytes** (*PyObject *o*)
Return value: New reference. Compute a bytes representation of object *o*. NULL is returned on failure and a bytes object on success. This is equivalent to the Python expression bytes(o), when *o* is not an integer. Unlike bytes(o), a TypeError is raised when *o* is an integer instead of a zero-initialized bytes object.

int **PyObject_IsSubclass** (*PyObject *derived*, *PyObject *cls*)
Return 1 if the class *derived* is identical to or derived from the class *cls*, otherwise return 0. In case of an error, return -1.

If *cls* is a tuple, the check will be done against every entry in *cls*. The result will be 1 when at least one of the checks returns 1, otherwise it will be 0.

If *cls* has a __subclasscheck__() method, it will be called to determine the subclass status as described in **PEP 3119**. Otherwise, *derived* is a subclass of *cls* if it is a direct or indirect subclass, i.e. contained in cls.__mro__.

Normally only class objects, i.e. instances of type or a derived class, are considered classes. However, objects can override this by having a __bases__ attribute (which must be a tuple of base classes).

int **PyObject_IsInstance** (*PyObject *inst*, *PyObject *cls*)
Return 1 if *inst* is an instance of the class *cls* or a subclass of *cls*, or 0 if not. On error, returns -1 and sets an exception.

If *cls* is a tuple, the check will be done against every entry in *cls*. The result will be 1 when at least one of the checks returns 1, otherwise it will be 0.

If *cls* has a __instancecheck__() method, it will be called to determine the subclass status as described in **PEP 3119**. Otherwise, *inst* is an instance of *cls* if its class is a subclass of *cls*.

An instance *inst* can override what is considered its class by having a __class__ attribute.

An object *cls* can override if it is considered a class, and what its base classes are, by having a __bases__ attribute (which must be a tuple of base classes).

int **PyCallable_Check** (*PyObject *o*)
Determine if the object *o* is callable. Return 1 if the object is callable and 0 otherwise. This function always succeeds.

*PyObject** **PyObject_Call**(*PyObject *callable*, *PyObject *args*, *PyObject *kwargs*)
Return value: New reference. Call a callable Python object *callable*, with arguments given by the tuple *args*, and named arguments given by the dictionary *kwargs*.

args must not be `NULL`, use an empty tuple if no arguments are needed. If no named arguments are needed, *kwargs* can be `NULL`.

Return the result of the call on success, or raise an exception and return `NULL` on failure.

This is the equivalent of the Python expression: `callable(*args, **kwargs)`.

*PyObject** **PyObject_CallObject**(*PyObject *callable*, *PyObject *args*)
Return value: New reference. Call a callable Python object *callable*, with arguments given by the tuple *args*. If no arguments are needed, then *args* can be `NULL`.

Return the result of the call on success, or raise an exception and return `NULL` on failure.

This is the equivalent of the Python expression: `callable(*args)`.

*PyObject** **PyObject_CallFunction**(*PyObject *callable*, const char *format*, ...)
Return value: New reference. Call a callable Python object *callable*, with a variable number of C arguments. The C arguments are described using a *Py_BuildValue()* style format string. The format can be `NULL`, indicating that no arguments are provided.

Return the result of the call on success, or raise an exception and return `NULL` on failure.

This is the equivalent of the Python expression: `callable(*args)`.

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

버전 3.4에서 변경: The type of *format* was changed from `char *`.

*PyObject** **PyObject_CallMethod**(*PyObject *obj*, const char *name*, const char *format*, ...)
Return value: New reference. Call the method named *name* of object *obj* with a variable number of C arguments. The C arguments are described by a *Py_BuildValue()* format string that should produce a tuple.

The format can be `NULL`, indicating that no arguments are provided.

Return the result of the call on success, or raise an exception and return `NULL` on failure.

This is the equivalent of the Python expression: `obj.name(arg1, arg2, ...)`.

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

버전 3.4에서 변경: The types of *name* and *format* were changed from `char *`.

*PyObject** **PyObject_CallFunctionObjArgs**(*PyObject *callable*, ..., NULL)
Return value: New reference. Call a callable Python object *callable*, with a variable number of *PyObject** arguments. The arguments are provided as a variable number of parameters followed by `NULL`.

Return the result of the call on success, or raise an exception and return `NULL` on failure.

This is the equivalent of the Python expression: `callable(arg1, arg2, ...)`.

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

Return the result of the call on success, or raise an exception and return `NULL` on failure.

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

버전 3.2에서 변경: The return type is now Py_hash_t. This is a signed integer the same size as Py_ssize_t.

Py_hash_t **PyObject_HashNotImplemented**(*PyObject* *o)
Set a TypeError indicating that type(o) is not hashable and return -1. This function receives special treatment when stored in a tp_hash slot, allowing a type to explicitly indicate to the interpreter that it is not hashable.

int **PyObject_IsTrue**(*PyObject* *o)
Returns 1 if the object *o* is considered to be true, and 0 otherwise. This is equivalent to the Python expression not not o. On failure, return -1.

int **PyObject_Not**(*PyObject* *o)
Returns 0 if the object *o* is considered to be true, and 1 otherwise. This is equivalent to the Python expression not o. On failure, return -1.

*PyObject** **PyObject_Type**(*PyObject* *o)
Return value: New reference. When *o* is non-NULL, returns a type object corresponding to the object type of object *o*. On failure, raises SystemError and returns NULL. This is equivalent to the Python expression type(o). This function increments the reference count of the return value. There's really no reason to use this function instead of the common expression o->ob_type, which returns a pointer of type *PyTypeObject**, except when the incremented reference count is needed.

int **PyObject_TypeCheck**(*PyObject* *o, *PyTypeObject* *type)
Return true if the object *o* is of type *type* or a subtype of *type*. Both parameters must be non-NULL.

Py_ssize_t **PyObject_Size**(*PyObject* *o)
Py_ssize_t **PyObject_Length**(*PyObject* *o)
Return the length of object *o*. If the object *o* provides either the sequence and mapping protocols, the sequence length is returned. On error, -1 is returned. This is the equivalent to the Python expression len(o).

Py_ssize_t **PyObject_LengthHint**(*PyObject* *o, Py_ssize_t *default*)
Return an estimated length for the object *o*. First try to return its actual length, then an estimate using __length_hint__(), and finally return the default value. On error return -1. This is the equivalent to the Python expression operator.length_hint(o, default).

버전 3.4에 추가.

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

int **PyObject_SetItem**(*PyObject* *o, *PyObject* *key, *PyObject* *v)
Map the object *key* to the value *v*. Raise an exception and return -1 on failure; return 0 on success. This is the equivalent of the Python statement o[key] = v.

int **PyObject_DelItem**(*PyObject* *o, *PyObject* *key)
Remove the mapping for the object *key* from the object *o*. Return -1 on failure. This is equivalent to the Python statement del o[key].

*PyObject** **PyObject_Dir**(*PyObject* *o)
Return value: New reference. This is equivalent to the Python expression dir(o), returning a (possibly empty) list of strings appropriate for the object argument, or NULL if there was an error. If the argument is NULL, this is like the Python dir(), returning the names of the current locals; in this case, if no execution frame is active then NULL is returned but *PyErr_Occurred()* will return false.

*PyObject** **PyObject_GetIter**(*PyObject* *o)
Return value: New reference. This is equivalent to the Python expression iter(o). It returns a new iterator for the object argument, or the object itself if the object is already an iterator. Raises TypeError and returns NULL if the object cannot be iterated.

7.2 숫자 프로토콜

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

*PyObject** **PyNumber_Add**(*PyObject* *o1, *PyObject* *o2)
Return value: New reference. Returns the result of adding *o1* and *o2*, or NULL on failure. This is the equivalent of the Python expression `o1 + o2`.

*PyObject** **PyNumber_Subtract**(*PyObject* *o1, *PyObject* *o2)
Return value: New reference. Returns the result of subtracting *o2* from *o1*, or NULL on failure. This is the equivalent of the Python expression `o1 - o2`.

*PyObject** **PyNumber_Multiply**(*PyObject* *o1, *PyObject* *o2)
Return value: New reference. Returns the result of multiplying *o1* and *o2*, or NULL on failure. This is the equivalent of the Python expression `o1 * o2`.

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

버전 3.5에 추가.

*PyObject** **PyNumber_FloorDivide**(*PyObject* *o1, *PyObject* *o2)
Return value: New reference. Return the floor of *o1* divided by *o2*, or NULL on failure. This is equivalent to the "classic" division of integers.

*PyObject** **PyNumber_TrueDivide**(*PyObject* *o1, *PyObject* *o2)
Return value: New reference. Return a reasonable approximation for the mathematical value of *o1* divided by *o2*, or NULL on failure. The return value is "approximate" because binary floating point numbers are approximate; it is not possible to represent all real numbers in base two. This function can return a floating point value when passed two integers.

*PyObject** **PyNumber_Remainder**(*PyObject* *o1, *PyObject* *o2)
Return value: New reference. Returns the remainder of dividing *o1* by *o2*, or NULL on failure. This is the equivalent of the Python expression `o1 % o2`.

*PyObject** **PyNumber_Divmod**(*PyObject* *o1, *PyObject* *o2)
Return value: New reference. See the built-in function `divmod()`. Returns NULL on failure. This is the equivalent of the Python expression `divmod(o1, o2)`.

*PyObject** **PyNumber_Power**(*PyObject* *o1, *PyObject* *o2, *PyObject* *o3)
Return value: New reference. See the built-in function `pow()`. Returns NULL on failure. This is the equivalent of the Python expression `pow(o1, o2, o3)`, where *o3* is optional. If *o3* is to be ignored, pass *Py_None* in its place (passing NULL for *o3* would cause an illegal memory access).

*PyObject** **PyNumber_Negative**(*PyObject* *o)
Return value: New reference. Returns the negation of *o* on success, or NULL on failure. This is the equivalent of the Python expression `-o`.

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

*PyObject** **PyNumber_Absolute**(*PyObject* *o)
Return value: New reference. Returns the absolute value of *o*, or NULL on failure. This is the equivalent of the Python expression `abs(o)`.

*PyObject** **PyNumber_Invert**(*PyObject* *o)
Return value: New reference. Returns the bitwise negation of *o* on success, or NULL on failure. This is the equivalent of the Python expression `~o`.

*PyObject** **PyNumber_Lshift**(*PyObject *o1*, *PyObject *o2*)
Return value: New reference. Returns the result of left shifting *o1* by *o2* on success, or NULL on failure. This is the equivalent of the Python expression `o1 << o2`.

*PyObject** **PyNumber_Rshift**(*PyObject *o1*, *PyObject *o2*)
Return value: New reference. Returns the result of right shifting *o1* by *o2* on success, or NULL on failure. This is the equivalent of the Python expression `o1 >> o2`.

*PyObject** **PyNumber_And**(*PyObject *o1*, *PyObject *o2*)
Return value: New reference. Returns the "bitwise and" of *o1* and *o2* on success and NULL on failure. This is the equivalent of the Python expression `o1 & o2`.

*PyObject** **PyNumber_Xor**(*PyObject *o1*, *PyObject *o2*)
Return value: New reference. Returns the "bitwise exclusive or" of *o1* by *o2* on success, or NULL on failure. This is the equivalent of the Python expression `o1 ^ o2`.

*PyObject** **PyNumber_Or**(*PyObject *o1*, *PyObject *o2*)
Return value: New reference. Returns the "bitwise or" of *o1* and *o2* on success, or NULL on failure. This is the equivalent of the Python expression `o1 | o2`.

*PyObject** **PyNumber_InPlaceAdd**(*PyObject *o1*, *PyObject *o2*)
Return value: New reference. Returns the result of adding *o1* and *o2*, or NULL on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 += o2`.

*PyObject** **PyNumber_InPlaceSubtract**(*PyObject *o1*, *PyObject *o2*)
Return value: New reference. Returns the result of subtracting *o2* from *o1*, or NULL on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 -= o2`.

*PyObject** **PyNumber_InPlaceMultiply**(*PyObject *o1*, *PyObject *o2*)
Return value: New reference. Returns the result of multiplying *o1* and *o2*, or NULL on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 *= o2`.

*PyObject** **PyNumber_InPlaceMatrixMultiply**(*PyObject *o1*, *PyObject *o2*)
Return value: New reference. Returns the result of matrix multiplication on *o1* and *o2*, or NULL on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 @= o2`.

버전 3.5에 추가.

*PyObject** **PyNumber_InPlaceFloorDivide**(*PyObject *o1*, *PyObject *o2*)
Return value: New reference. Returns the mathematical floor of dividing *o1* by *o2*, or NULL on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 //= o2`.

*PyObject** **PyNumber_InPlaceTrueDivide**(*PyObject *o1*, *PyObject *o2*)
Return value: New reference. Return a reasonable approximation for the mathematical value of *o1* divided by *o2*, or NULL on failure. The return value is "approximate" because binary floating point numbers are approximate; it is not possible to represent all real numbers in base two. This function can return a floating point value when passed two integers. The operation is done *in-place* when *o1* supports it.

*PyObject** **PyNumber_InPlaceRemainder**(*PyObject *o1*, *PyObject *o2*)
Return value: New reference. Returns the remainder of dividing *o1* by *o2*, or NULL on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 %= o2`.

*PyObject** **PyNumber_InPlacePower**(*PyObject *o1*, *PyObject *o2*, *PyObject *o3*)
Return value: New reference. See the built-in function `pow()`. Returns NULL on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 **= o2` when o3 is *Py_None*, or an in-place variant of `pow(o1, o2, o3)` otherwise. If *o3* is to be ignored, pass *Py_None* in its place (passing NULL for *o3* would cause an illegal memory access).

*PyObject** **PyNumber_InPlaceLshift**(*PyObject *o1*, *PyObject *o2*)
Return value: New reference. Returns the result of left shifting *o1* by *o2* on success, or NULL on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 <<= o2`.

*PyObject** **PyNumber_InPlaceRshift**(*PyObject *o1*, *PyObject *o2*)
Return value: New reference. Returns the result of right shifting *o1* by *o2* on success, or NULL on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 >>= o2`.

*PyObject** **PyNumber_InPlaceAnd** (*PyObject *o1*, *PyObject *o2*)
Return value: New reference. Returns the "bitwise and" of *o1* and *o2* on success and NULL on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 &= o2`.

*PyObject** **PyNumber_InPlaceXor** (*PyObject *o1*, *PyObject *o2*)
Return value: New reference. Returns the "bitwise exclusive or" of *o1* by *o2* on success, or NULL on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 ^= o2`.

*PyObject** **PyNumber_InPlaceOr** (*PyObject *o1*, *PyObject *o2*)
Return value: New reference. Returns the "bitwise or" of *o1* and *o2* on success, or NULL on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 |= o2`.

*PyObject** **PyNumber_Long** (*PyObject *o*)
Return value: New reference. Returns the *o* converted to an integer object on success, or NULL on failure. This is the equivalent of the Python expression `int(o)`.

*PyObject** **PyNumber_Float** (*PyObject *o*)
Return value: New reference. Returns the *o* converted to a float object on success, or NULL on failure. This is the equivalent of the Python expression `float(o)`.

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

*PyObject** **PyNumber_ToBase** (*PyObject *n*, int *base*)
Return value: New reference. 정수 *n*을 진수 *base*를 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. *n*이 파이썬 int가 아니면, 먼저 *PyNumber_Index()*로 변환됩니다.

Py_ssize_t **PyNumber_AsSsize_t** (*PyObject *o*, *PyObject *exc*)
*o*가 정수로 해석될 수 있으면, *o*를 Py_ssize_t 값으로 변환하여 반환합니다. 호출이 실패하면, 예외가 발생하고 `-1`이 반환됩니다.

If *o* can be converted to a Python int but the attempt to convert to a Py_ssize_t value would raise an `OverflowError`, then the *exc* argument is the type of exception that will be raised (usually `IndexError` or `OverflowError`). If *exc* is NULL, then the exception is cleared and the value is clipped to `PY_SSIZE_T_MIN` for a negative integer or `PY_SSIZE_T_MAX` for a positive integer.

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

7.3 시퀀스 프로토콜

int **PySequence_Check** (*PyObject *o*)
객체가 시퀀스 프로토콜을 제공하면 `1`을 반환하고, 그렇지 않으면 `0`을 반환합니다. `__getitem__()` 메서드가 있는 파이썬 클래스의 경우 `dict` 서브 클래스가 아닌 한 `1`을 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 `-1`을 반환합니다. 이것은 파이썬 표현식 `len(o)`와 동등합니다.

*PyObject** **PySequence_Concat** (*PyObject *o1*, *PyObject *o2*)
Return value: New reference. Return the concatenation of *o1* and *o2* on success, and NULL on failure. This is the equivalent of the Python expression `o1 + o2`.

*PyObject** **PySequence_Repeat** (*PyObject *o*, Py_ssize_t *count*)
Return value: New reference. Return the result of repeating sequence object *o* *count* times, or NULL on failure. This is the equivalent of the Python expression `o * count`.

*PyObject** **PySequence_InPlaceConcat** (*PyObject *o1*, *PyObject *o2*)
Return value: New reference. Return the concatenation of *o1* and *o2* on success, and `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python expression `o1 += o2`.

*PyObject** **PySequence_InPlaceRepeat** (*PyObject *o*, Py_ssize_t *count*)
Return value: New reference. Return the result of repeating sequence object *o* *count* times, or `NULL` on failure. The operation is done *in-place* when *o* supports it. This is the equivalent of the Python expression `o *= count`.

*PyObject** **PySequence_GetItem** (*PyObject *o*, Py_ssize_t *i*)
Return value: New reference. Return the *i*th element of *o*, or `NULL` on failure. This is the equivalent of the Python expression `o[i]`.

*PyObject** **PySequence_GetSlice** (*PyObject *o*, Py_ssize_t *i1*, Py_ssize_t *i2*)
Return value: New reference. Return the slice of sequence object *o* between *i1* and *i2*, or `NULL` on failure. This is the equivalent of the Python expression `o[i1:i2]`.

int **PySequence_SetItem** (*PyObject *o*, Py_ssize_t *i*, *PyObject *v*)
객체 *v*를 *o*의 *i* 번째 요소에 대입합니다. 실패하면 예외를 발생시키고 `-1`을 반환합니다; 성공하면 `0`을 반환합니다. 이것은 파이썬 문장 `o[i] = v`와 동등합니다. 이 함수는 *v*에 대한 참조를 훔치지 않습니다.

If *v* is `NULL`, the element is deleted, however this feature is deprecated in favour of using *PySequence_DelItem()*.

int **PySequence_DelItem** (*PyObject *o*, Py_ssize_t *i*)
o 객체의 *i* 번째 요소를 삭제합니다. 실패하면 `-1`을 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 `-1`을 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 key의 수를 반환합니다. 실패하면 `-1`을 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 `1`을 반환하고, 그렇지 않으면 `0`을 반환합니다. 에러 시 `-1`을 반환합니다. 이는 파이썬 표현식 `value in o`와 동등합니다.

Py_ssize_t **PySequence_Index** (*PyObject *o*, *PyObject *value*)
`o[i] == value`을 만족하는 첫 번째 인덱스 *i*를 반환합니다. 에러 시 `-1`을 반환합니다. 이것은 파이썬 표현식 `o.index(value)`와 동등합니다.

*PyObject** **PySequence_List** (*PyObject *o*)
Return value: New reference. Return a list object with the same contents as the sequence or iterable *o*, or `NULL` on failure. The returned list is guaranteed to be new. This is equivalent to the Python expression `list(o)`.

*PyObject** **PySequence_Tuple** (*PyObject *o*)
Return value: New reference. Return a tuple object with the same contents as the sequence or iterable *o*, or `NULL` on failure. If *o* is a tuple, a new reference will be returned, otherwise a tuple will be constructed with the appropriate contents. This is equivalent to the Python expression `tuple(o)`.

*PyObject** **PySequence_Fast** (*PyObject *o*, const char *m*)
Return value: New reference. Return the sequence or iterable *o* as an object usable by the other `PySequence_Fast*` family of functions. If the object is not a sequence or iterable, raises `TypeError` with *m* as the message text. Returns `NULL` on failure.

The `PySequence_Fast*` functions are thus named because they assume *o* is a *PyTupleObject* or a *PyListObject* and access the data fields of *o* directly.

As a CPython implementation detail, if *o* is already a sequence or list, it will be returned.

Py_ssize_t **PySequence_Fast_GET_SIZE**(*PyObject* *o)
Returns the length of *o*, assuming that *o* was returned by `PySequence_Fast()` and that *o* is not `NULL`. The size can also be gotten by calling `PySequence_Size()` on *o*, but `PySequence_Fast_GET_SIZE()` is faster because it can assume *o* is a list or tuple.

*PyObject** **PySequence_Fast_GET_ITEM**(*PyObject* *o, Py_ssize_t *i*)
Return value: Borrowed reference. Return the *i*th element of *o*, assuming that *o* was returned by `PySequence_Fast()`, *o* is not `NULL`, and that *i* is within bounds.

*PyObject*** **PySequence_Fast_ITEMS**(*PyObject* *o)
Return the underlying array of PyObject pointers. Assumes that *o* was returned by `PySequence_Fast()` and *o* is not `NULL`.

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. Return the *i*th element of *o* or `NULL` on failure. Faster form of `PySequence_GetItem()` but without checking that `PySequence_Check()` on *o* is true and without adjustment for negative indices.

7.4 매핑 프로토콜

`PyObject_GetItem()`, `PyObject_SetItem()` 및 `PyObject_DelItem()`도 참조하십시오.

int **PyMapping_Check**(*PyObject* *o)
Return `1` if the object provides mapping protocol or supports slicing, and `0` otherwise. Note that it returns `1` for Python classes with a `__getitem__()` method since in general case it is impossible to determine what type of keys it supports. This function always succeeds.

Py_ssize_t **PyMapping_Size**(*PyObject* *o)
Py_ssize_t **PyMapping_Length**(*PyObject* *o)
성공 시 객체 *o*의 키 수를 반환하고, 실패하면 `-1`을 반환합니다. 이는 파이썬 표현식 `len(o)`와 동등합니다.

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

int **PyMapping_SetItemString**(*PyObject* *o, const char *key, *PyObject* *v)
객체 *o*에서 문자열 *key*를 값 *v*에 매핑합니다. 실패하면 `-1`을 반환합니다. 이것은 파이썬 문장 `o[key] = v`와 동등합니다. `PyObject_SetItem()`도 참조하십시오.

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

int **PyMapping_HasKey**(*PyObject* *o, *PyObject* *key)
매핑 객체에 *key* 키가 있으면 `1`을 반환하고, 그렇지 않으면 `0`을 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 `1`을 반환하고, 그렇지 않으면 `0`을 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. On success, return a list of the keys in object *o*. On failure, return NULL.

버전 3.7에서 변경: 이전에는 함수가 리스트나 튜플을 반환했습니다.

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

버전 3.7에서 변경: 이전에는 함수가 리스트나 튜플을 반환했습니다.

*PyObject** **PyMapping_Items**(*PyObject *o*)
Return value: New reference. On success, return a list of the items in object *o*, where each item is a tuple containing a key-value pair. On failure, return NULL.

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. Return the next value from the iteration *o*. The object must be an iterator (it is up to the caller to check this). If there are no remaining values, returns NULL with no exception set. If an error occurs while retrieving the item, returns NULL and passes along the exception.

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 *Buffer Object Structures* 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**
A new reference to the exporting object. The reference is owned by the consumer and automatically decremented and set to NULL by *PyBuffer_Release()*. The field is the equivalent of the return value of any standard C-API function.

As a special case, for *temporary* buffers that are wrapped by *PyMemoryView_FromBuffer()* or *PyBuffer_FillInfo()* this field is NULL. In general, exporting objects MUST NOT use this scheme.

Py_ssize_t **len**
product(shape) * itemsize. 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**
Item size in bytes of a single element. Same as the value of struct.calcsize() called on non-NULL *format* values.

Important exception: If a consumer requests a buffer without the *PyBUF_FORMAT* flag, *format* will be set to NULL, but *itemsize* still has the value for the original format.

*shape*이 있으면, product(shape) * itemsize == len 동치가 계속 성립하고 소비자는 *itemsize*를 사용하여 버퍼를 탐색할 수 있습니다.

If *shape* is NULL as a result of a *PyBUF_SIMPLE* or a *PyBUF_WRITABLE* request, the consumer must disregard *itemsize* and assume itemsize == 1.

const char ***format**
A *NUL* terminated string in struct module style syntax describing the contents of a single item. If this is NULL, "B" (unsigned bytes) is assumed.

이 필드는 *PyBUF_FORMAT* 플래그로 제어됩니다.

int **ndim**
The number of dimensions the memory represents as an n-dimensional array. If it is 0, *buf* points to a single item representing a scalar. In this case, *shape*, *strides* and *suboffsets* MUST be NULL.

매크로 PyBUF_MAX_NDIM는 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] * itemsize는 *len*과 같아야 합니다.

모양 값은 shape[n] >= 0로 제한됩니다. shape[n] == 0인 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] <= 0인 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).

If all suboffsets are negative (i.e. no de-referencing is needed), then this field must be `NULL` (the default value).

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
Controls the *format* field. If set, this field MUST be filled in correctly. Otherwise, this field MUST be `NULL`.

*PyBUF_WRITABLE*은 다음 섹션의 모든 플래그와 | 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 | 될 수 있습니다. PyBUF_SIMPLE은 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
PyBUF_ND	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: shape과 strides

NumPy 스타일 배열의 논리적 구조는 *itemsize*, *ndim*, *shape* 및 *strides*로 정의됩니다.

If `ndim == 0`, the memory location pointed to by *buf* is interpreted as a scalar of size *itemsize*. In that case, both *shape* and *strides* are `NULL`.

If *strides* is `NULL`, the array is interpreted as a standard n-dimensional C-array. Otherwise, the consumer must access an n-dimensional array as follows:

```
ptr = (char *)buf + indices[0] * strides[0] + ... + indices[n-1] * strides[n-1];
item = *((typeof(item) *)ptr);
```

위에서 언급했듯이, *buf*는 실제 메모리 블록 내의 모든 위치를 가리킬 수 있습니다. 제공자(exporter)는 이 함수로 버퍼의 유효성을 검사 할 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
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    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 임베드 될 수 있는데, 메모리의 어느 위치에나 배치될 수 있는 두 개의 `char x[2][3]` 배열을 가리킵니다.

Here is a function that returns a pointer to the element in an N-D array pointed to by an N-dimensional index when there are both non-`NULL` strides and suboffsets:

```
void *get_item_pointer(int ndim, void *buf, Py_ssize_t *strides,
                       Py_ssize_t *suboffsets, Py_ssize_t *indices) {
    char *pointer = (char*)buf;
    int i;
    for (i = 0; i < ndim; i++) {
        pointer += strides[i] * indices[i];
        if (suboffsets[i] >=0 ) {
            pointer = *((char**)pointer) + suboffsets[i];
        }
    }
    return (void*)pointer;
}
```

7.6.4 버퍼 관련 함수

int **PyObject_CheckBuffer**(*PyObject* *obj)
*obj*가 버퍼 인터페이스를 지원하면 `1`을 반환하고, 그렇지 않으면 `0`을 반환합니다. `1`이 반환될 때, *PyObject_GetBuffer()*가 성공할 것이라고 보장하지는 않습니다. 이 함수는 항상 성공합니다.

int **PyObject_GetBuffer**(*PyObject* *exporter, *Py_buffer* *view, int *flags*)
Send a request to *exporter* to fill in *view* as specified by *flags*. If the exporter cannot provide a buffer of the exact type, it MUST raise `PyExc_BufferError`, set `view->obj` to `NULL` and return `-1`.

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 *itemsize*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 `1`을 반환합니다. 그렇지 않으면 `0`을 반환합니다. 이 함수는 항상 성공합니다.

void* **PyBuffer_GetPointer**(*Py_buffer* *view, Py_ssize_t *indices)
Get the memory area pointed to by the *indices* inside the given *view*. *indices* must point to an array of `view->ndim` indices.

int **PyBuffer_FromContiguous**(*Py_buffer* *view, void *buf, Py_ssize_t len, char fort)
Copy contiguous *len* bytes from *buf* to *view*. *fort* can be `'C'` or `'F'` (for C-style or Fortran-style ordering). `0` is returned on success, `-1` on error.

int **PyBuffer_ToContiguous**(void *buf, *Py_buffer* *src, Py_ssize_t len, char order)
Copy *len* bytes from *src* to its contiguous representation in *buf*. *order* can be `'C'` or `'F'` or `'A'` (for C-style or Fortran-style ordering or either one). `0` is returned on success, `-1` on error.

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.

On success, set `view->obj` to a new reference to *exporter* and return 0. Otherwise, raise `PyExc_BufferError`, set `view->obj` to `NULL` and return `-1`;

If this function is used as part of a *getbufferproc*, *exporter* MUST be set to the exporting object and *flags* must be passed unmodified. Otherwise, *exporter* MUST be `NULL`.

7.7 낡은 버퍼 프로토콜

버전 3.0부터 폐지.

이 함수는 파이썬 2에서 "낡은 버퍼 프로토콜" API 일부분이었습니다. 파이썬 3에서는 이 프로토콜이 더는 존재하지 않지만 2.x 코드 이식을 쉽게 하도록 함수들은 여전히 노출됩니다. 이들은 새 버퍼 프로토콜를 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, `0`을 반환하고, *buffer*를 메모리 위치로 설정하고, *buffer_len*을 버퍼 길이로 설정합니다. 에러 시에, `-1`을 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, `0`을 반환하고, *buffer*를 메모리 위치로 설정하고, *buffer_len*을 버퍼 길이로 설정합니다. 에러 시에, `-1`을 반환하고, `TypeError`를 설정합니다.

int **PyObject_CheckReadBuffer**(*PyObject* *o)
*o*가 단일 세그먼트 읽기 가능 버퍼 인터페이스를 지원하면 `1`을 반환합니다. 그렇지 않으면, `0`을 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, `0`을 반환하고, *buffer*를 메모리 위치로 설정하고, *buffer_len*을 버퍼 길이로 설정합니다. 에러 시에, `-1`을 반환하고, `TypeError`를 설정합니다.

CHAPTER 8

구상 객체 계층

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: While the functions described in this chapter carefully check the type of the objects which are passed in, many of them do not check for `NULL` being passed instead of a valid object. Allowing `NULL` to be passed in can cause memory access violations and immediate termination of the interpreter.

8.1 기본 객체

이 절에서는 파이썬 형 객체와 싱글톤 객체 `None`에 대해 설명합니다.

8.1.1 형 객체

PyTypeObject
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*)
*type*의 *tp_flags* 멤버를 반환합니다. 이 함수는 주로 *Py_LIMITED_API*와 함께 사용하기 위한 것입

니다; 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 `long`이 아니라 `unsigned long`입니다.

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 *o*가 기능 *feature*를 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*)
*a*가 *b*의 서브 형이면 참을 반환합니다.

이 함수는 실제 서브 형만 검사합니다. 즉, `__subclasscheck__()`가 *b*에 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. Generic handler for the *tp_alloc* slot of a type object. Use Python's default memory allocation mechanism to allocate a new instance and initialize all its contents to `NULL`.

*PyObject** **PyType_GenericNew**(*PyTypeObject *type*, *PyObject *args*, *PyObject *kwds*)
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 개체를 마무리합니다. 초기화를 완료하려면 모든 형 객체에 대해 이 메서드를 호출해야 합니다. 이 함수는 형의 베이스 클래스에서 상속된 슬롯을 추가합니다. 성공 시 `0`을 반환하고, 오류 시 `-1`을 반환하고 예외를 설정합니다.

*PyObject** **PyType_FromSpec**(PyType_Spec *spec)
Return value: New reference. 함수에 전달된 *spec*으로 힙 형 객체를 만들고 반환합니다.

*PyObject** **PyType_FromSpecWithBases**(PyType_Spec *spec, *PyObject *bases*)
Return value: New reference. *spec*으로 힙 형 객체를 만들고 반환합니다. 이 외에도, 생성된 힙 형에는 *bases* 튜플에 포함된 모든 형이 베이스형으로 포함됩니다. 이를 통해 호출자는 다른 힙 형을 베이스형으로 참조할 수 있습니다.

버전 3.3에 추가.

void* **PyType_GetSlot**(*PyTypeObject *type*, int *slot*)
Return the function pointer stored in the given slot. If the result is `NULL`, this indicates that either the slot is `NULL`, or that the function was called with invalid parameters. Callers will typically cast the result pointer into the appropriate function type.

버전 3.4에 추가.

8.1.2 None 객체

None에 대한 *PyTypeObject*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.

에러 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.

PyTypeObject **PyLong_Type**
이 *PyTypeObject* 인스턴스는 파이썬 정수 형을 나타냅니다. 이것은 파이썬 계층의 int와 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. Return a new *PyLongObject* object from *v*, or NULL on failure.

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. Return a new *PyLongObject* object from a C unsigned long, or NULL on failure.

*PyObject** **PyLong_FromSsize_t**(Py_ssize_t *v*)
Return value: New reference. Return a new *PyLongObject* object from a C Py_ssize_t, or NULL on failure.

*PyObject** **PyLong_FromSize_t**(size_t *v*)
Return value: New reference. Return a new *PyLongObject* object from a C size_t, or NULL on failure.

*PyObject** **PyLong_FromLongLong**(long long *v*)
Return value: New reference. Return a new *PyLongObject* object from a C long long, or NULL on failure.

*PyObject** **PyLong_FromUnsignedLongLong**(unsigned long long *v*)
Return value: New reference. Return a new *PyLongObject* object from a C unsigned long long, or NULL on failure.

*PyObject** **PyLong_FromDouble** (double *v*)
Return value: New reference. Return a new `PyLongObject` object from the integer part of *v*, or `NULL` on failure.

*PyObject** **PyLong_FromString** (const char **str*, char ***pend*, int *base*)
Return value: New reference. Return a new `PyLongObject` based on the string value in *str*, which is interpreted according to the radix in *base*. If *pend* is non-`NULL`, **pend* will point to the first character in *str* which follows the representation of the number. If *base* is `0`, *str* is interpreted using the integers definition; in this case, leading zeros in a non-zero decimal number raises a `ValueError`. If *base* is not `0`, it must be between `2` and `36`, inclusive. Leading spaces and single underscores after a base specifier and between digits are ignored. If there are no digits, `ValueError` will be raised.

*PyObject** **PyLong_FromUnicode** (*Py_UNICODE* **u*, Py_ssize_t *length*, int *base*)
Return value: New reference. 유니코드 숫자의 시퀀스를 파이썬 정숫값으로 변환합니다. 유니코드 문자열은 먼저 `PyUnicode_EncodeDecimal()`을 사용하여 바이트열로 인코딩된 다음 `PyLong_FromString()`을 사용하여 변환됩니다.

Deprecated since version 3.3, will be removed in version 4.0: 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*의 유니코드 숫자 시퀀스를 파이썬 정숫값으로 변환합니다. 유니코드 문자열은 먼저 `PyUnicode_EncodeDecimal()`을 사용하여 바이트열로 인코딩된 다음 `PyLong_FromString()`을 사용하여 변환됩니다.

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 `__int__()` 메서드를 호출하여 `PyLongObject`로 변환합니다.

*obj*의 값이 `long`의 범위를 벗어나면 `OverflowError`를 발생시킵니다.

에러 시 `-1`을 반환합니다. 모호성을 제거하려면 `PyErr_Occurred()`를 사용하십시오.

long **PyLong_AsLongAndOverflow** (*PyObject* **obj*, int **overflow*)
*obj*의 C `long` 표현을 반환합니다. *obj*가 `PyLongObject`의 인스턴스가 아니면, (있다면) 먼저 `__int__()` 메서드를 호출하여 `PyLongObject`로 변환합니다.

*obj*의 값이 `LONG_MAX`보다 크거나 `LONG_MIN`보다 작으면, **overflow*를 각각 `1`이나 `-1`로 설정하고 `-1`을 반환합니다; 그렇지 않으면, **overflow*를 `0`으로 설정합니다. 다른 예외가 발생하면 **overflow*를 `0`으로 설정하고 `-1`을 평소와 같이 반환합니다.

에러 시 `-1`을 반환합니다. 모호성을 제거하려면 `PyErr_Occurred()`를 사용하십시오.

long long **PyLong_AsLongLong** (*PyObject* **obj*)
*obj*의 C `long long` 표현을 반환합니다. *obj*가 `PyLongObject`의 인스턴스가 아니면, (있다면) 먼저 `__int__()` 메서드를 호출하여 `PyLongObject`로 변환합니다.

Raise `OverflowError` if the value of *obj* is out of range for a `long long`.

에러 시 `-1`을 반환합니다. 모호성을 제거하려면 `PyErr_Occurred()`를 사용하십시오.

long long **PyLong_AsLongLongAndOverflow** (*PyObject* **obj*, int **overflow*)
*obj*의 C `long long` 표현을 반환합니다. *obj*가 `PyLongObject`의 인스턴스가 아니면, (있다면) 먼저 `__int__()` 메서드를 호출하여 `PyLongObject`로 변환합니다.

*obj*의 값이 `PY_LLONG_MAX`보다 크거나 `PY_LLONG_MIN`보다 작으면, **overflow*를 각각 `1`이나 `-1`로 설정하고 `-1`을 반환합니다; 그렇지 않으면, **overflow*를 `0`으로 설정합니다. 다른 예외가 발생하면 **overflow*를 `0`으로 설정하고 `-1`을 평소와 같이 반환합니다.

에러 시 `-1`을 반환합니다. 모호성을 제거하려면 `PyErr_Occurred()`를 사용하십시오.

버전 3.2에 추가.

Py_ssize_t **PyLong_AsSsize_t**(*PyObject *pylong*)
*pylong*의 C `Py_ssize_t` 표현을 반환합니다. *pylong*은 *PyLongObject*의 인스턴스여야 합니다.

*pylong*의 값이 `Py_ssize_t`의 범위를 벗어나면 `OverflowError`를 발생시킵니다.

에러 시 `-1`을 반환합니다. 모호성을 제거하려면 *PyErr_Occurred()*를 사용하십시오.

unsigned long **PyLong_AsUnsignedLong**(*PyObject *pylong*)
*pylong*의 C `unsigned long` 표현을 반환합니다. *pylong*은 *PyLongObject*의 인스턴스여야 합니다.

*pylong*의 값이 `unsigned long`의 범위를 벗어나면 `OverflowError`를 발생시킵니다.

에러 시 `(unsigned long)-1`을 반환합니다. 모호성을 제거하려면 *PyErr_Occurred()*를 사용하십시오.

size_t **PyLong_AsSize_t**(*PyObject *pylong*)
*pylong*의 C `size_t` 표현을 반환합니다. *pylong*은 *PyLongObject*의 인스턴스여야 합니다.

*pylong*의 값이 `size_t`의 범위를 벗어나면 `OverflowError`를 발생시킵니다.

에러 시 `(size_t)-1`을 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 long`의 범위를 벗어나면 `OverflowError`를 발생시킵니다.

에러 시 `(unsigned long long)-1`을 반환합니다. 모호성을 제거하려면 *PyErr_Occurred()*를 사용하십시오.

버전 3.1에서 변경: 음의 *pylong*는 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 `__int__()` 메서드를 호출하여 *PyLongObject*로 변환합니다.

*obj*의 값이 `unsigned long`의 범위를 벗어나면, 그 값의 모듈로 `ULONG_MAX + 1` 환원을 반환합니다.

에러 시 `(unsigned long)-1`을 반환합니다. 모호성을 제거하려면 *PyErr_Occurred()*를 사용하십시오.

unsigned long long **PyLong_AsUnsignedLongLongMask**(*PyObject *obj*)
*obj*의 C `unsigned long long` 표현을 반환합니다. *obj*가 *PyLongObject*의 인스턴스가 아니면, (있다면) 먼저 `__int__()` 메서드를 호출하여 *PyLongObject*로 변환합니다.

*obj*의 값이 `unsigned long long`의 범위를 벗어나면, 그 값의 모듈로 `PY_ULLONG_MAX + 1` 환원을 반환합니다.

에러 시 `(unsigned long long)-1`을 반환합니다. 모호성을 제거하려면 *PyErr_Occurred()*를 사용하십시오.

double **PyLong_AsDouble**(*PyObject *pylong*)
*pylong*의 C `double` 표현을 반환합니다. *pylong*은 *PyLongObject*의 인스턴스여야 합니다.

*pylong*의 값이 `double`의 범위를 벗어나면 `OverflowError`를 발생시킵니다.

에러 시 `-1.0`을 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.

Returns `NULL` on error. Use *PyErr_Occurred()* to disambiguate.

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 카운트와 관련해서는 다른 객체와 마찬가지로 처리해야 합니다.

*PyObject** **Py_True**
파이썬 `True` 객체. 이 객체는 메서드가 없습니다. 참조 카운트와 관련해서는 다른 객체와 마찬가지로 처리해야 합니다.

Py_RETURN_FALSE
함수에서 `Py_False`를 반환하고, 참조 카운트를 적절하게 증가시킵니다.

Py_RETURN_TRUE
함수에서 `Py_True`를 반환하고, 참조 카운트를 적절하게 증가시킵니다.

*PyObject** **PyBool_FromLong**(long *v*)
Return value: New reference. *v*의 논리값에 따라 `Py_True` 나 `Py_False`에 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 `float`와 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. Create a *PyFloatObject* object based on the string value in *str*, or `NULL` on failure.

*PyObject** **PyFloat_FromDouble**(double *v*)
Return value: New reference. Create a *PyFloatObject* object from *v*, or `NULL` on failure.

double **PyFloat_AsDouble**(*PyObject* *pyfloat)
pyfloat 의 내용의 C `double` 표현을 반환합니다. *pyfloat*가 파이썬 부동 소수점 객체가 아니지만 `__float__()` 메서드가 있으면, *pyfloat*를 float로 변환하기 위해 이 메서드가 먼저 호출됩니다. 이 메서드는 실패하면 `-1.0`을 반환하므로, *PyErr_Occurred()*를 호출하여 에러를 확인해야 합니다.

double **PyFloat_AS_DOUBLE**(*PyObject* *pyfloat)
에러 검사 없이 *pyfloat* 의 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.h`를 감싸는 얇은 래퍼입니다.

double **PyFloat_GetMax**()
최대 표현 가능한 유한 float *DBL_MAX*를 C `double`로 반환합니다.

double **PyFloat_GetMin**()
최소 정규화된(normalized) 양의 float *DBL_MIN*를 C `double`로 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 `EDOM`으로 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 `EDOM`으로 설정합니다.

파이썬 객체로서의 복소수

PyComplexObject
파이썬 복소수 객체를 나타내는 *PyObject*의 서브 형.

PyTypeObject **PyComplex_Type**
이 *PyTypeObject* 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. 실패하면, 이 메서드는 `-1.0`를 실숫값으로 반환합니다.

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. Return a new bytes object with a copy of the string *v* as value on success, and `NULL` on failure. The parameter *v* must not be `NULL`; it will not be checked.

*PyObject** **PyBytes_FromStringAndSize**(const char **v*, Py_ssize_t *len*)
Return value: New reference. Return a new bytes object with a copy of the string *v* as value and length *len* on success, and `NULL` on failure. If *v* is `NULL`, the contents of the bytes object are uninitialized.

*PyObject** **PyBytes_FromFormat**(const char **format*, ...)
Return value: New reference. C `printf()`-스타일 *format* 문자열과 가변 개수의 인자를 받아서, 결과 파이썬 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 vargs)
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 *o*의 바이트열 표현을 반환합니다.

Py_ssize_t **PyBytes_Size**(*PyObject* *o)
바이트열 객체 *o*의 길이를 반환합니다.

Py_ssize_t **PyBytes_GET_SIZE**(*PyObject* *o)
에러 검사 없는 *PyBytes_Size()*의 매크로 형식.

char* **PyBytes_AsString**(*PyObject* *o)
Return a pointer to the contents of *o*. The pointer refers to the internal buffer of *o*, which consists of len(o) + 1 bytes. The last byte in the buffer is always null, regardless of whether there are any other null bytes. The data must not be modified in any way, unless the object was just created using PyBytes_FromStringAndSize(NULL, size). It must not be deallocated. If *o* is not a bytes object at all, *PyBytes_AsString()* returns NULL and raises TypeError.

char* **PyBytes_AS_STRING**(*PyObject* *string)
에러 검사 없는 *PyBytes_AsString()*의 매크로 형식.

int **PyBytes_AsStringAndSize**(*PyObject* *obj, char **buffer, Py_ssize_t *length)
출력 변수 *buffer*와 *length*로 객체 *obj*의 널-종료 내용을 반환합니다.

If *length* is NULL, the bytes object may not contain embedded null bytes; if it does, the function returns −1 and a ValueError is raised.

buffer는 *obj*의 내부 버퍼를 가리키게 되는데, 끝에 추가 널 바이트가 포함됩니다(*length*에는 포함되지 않습니다). 객체가 PyBytes_FromStringAndSize(NULL, size)를 사용하여 방금 만들어진 경우가 아니면 데이터를 수정해서는 안 됩니다. 할당을 해제해서는 안 됩니다. *obj*가 바이트열 객체가 아니면 *PyBytes_AsStringAndSize()*는 -1을 반환하고 TypeError를 발생시킵니다.

버전 3.5에서 변경: 이전에는, 바이트열 객체에 널 바이트가 포함되어 있으면 TypeError가 발생했습니다.

void **PyBytes_Concat**(*PyObject* **bytes, *PyObject* *newpart)
Create a new bytes object in *bytes containing the contents of *newpart* appended to *bytes*; the caller will own the new reference. The reference to the old value of *bytes* will be stolen. If the new object cannot be created, the old reference to *bytes* will still be discarded and the value of *bytes will be set to NULL; the appropriate exception will be set.

[1] 정수 지정자 (d, u, ld, lu, zd, zu, i, x)에서: 0-변환 플래그는 정밀도를 지정해도 영향을 미칩니다.

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*)
A way to resize a bytes object even though it is "immutable". Only use this to build up a brand new bytes object; don't use this if the bytes may already be known in other parts of the code. It is an error to call this function if the refcount on the input bytes object is not one. Pass the address of an existing bytes object as an lvalue (it may be written into), and the new size desired. On success, **bytes* holds the resized bytes object and `0` is returned; the address in **bytes* may differ from its input value. If the reallocation fails, the original bytes object at **bytes* is deallocated, **bytes* is set to `NULL`, `MemoryError` is set, and `-1` is returned.

8.3.2 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 `bytearray`와 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. Create a new bytearray object from *string* and its length, *len*. On failure, `NULL` is returned.

*PyObject** **PyByteArray_Concat**(*PyObject* **a*, *PyObject* **b*)
Return value: New reference. 바이트 배열 *a* 와 *b*를 이어붙여 새로운 bytearray로 반환합니다.

Py_ssize_t **PyByteArray_Size**(*PyObject* **bytearray*)
Return the size of *bytearray* after checking for a `NULL` pointer.

char* **PyByteArray_AsString**(*PyObject* **bytearray*)
Return the contents of *bytearray* as a char array after checking for a `NULL` pointer. The returned array always has an extra null byte appended.

int **PyByteArray_Resize**(*PyObject* **bytearray*, Py_ssize_t *len*)
*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 Unicode Objects and Codecs

Unicode Objects

Since the implementation of **PEP 393** in Python 3.3, Unicode objects internally use a variety of representations, in order to allow handling the complete range of Unicode characters while staying memory efficient. There are special cases for strings where all code points are below 128, 256, or 65536; otherwise, code points must be below 1114112 (which is the full Unicode range).

*Py_UNICODE** and UTF-8 representations are created on demand and cached in the Unicode object. The *Py_UNICODE** representation is deprecated and inefficient; it should be avoided in performance- or memory-sensitive situations.

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

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

Unicode Type

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

Py_UCS4
Py_UCS2
Py_UCS1
These types are typedefs for unsigned integer types wide enough to contain characters of 32 bits, 16 bits and 8 bits, respectively. When dealing with single Unicode characters, use *Py_UCS4*.

버전 3.3에 추가.

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

버전 3.3에서 변경: In previous versions, this was a 16-bit type or a 32-bit type depending on whether you selected a "narrow" or "wide" Unicode version of Python at build time.

PyASCIIObject
PyCompactUnicodeObject
PyUnicodeObject
These subtypes of *PyObject* represent a Python Unicode object. In almost all cases, they shouldn't be used directly, since all API functions that deal with Unicode objects take and return *PyObject* pointers.

버전 3.3에 추가.

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

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

int **PyUnicode_Check**(*PyObject* *o)
Return true if the object *o* is a Unicode object or an instance of a Unicode subtype.

int **PyUnicode_CheckExact**(*PyObject* *o)
Return true if the object *o* is a Unicode object, but not an instance of a subtype.

int **PyUnicode_READY**(*PyObject* *o)
Ensure the string object *o* is in the "canonical" representation. This is required before using any of the access macros described below.

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

버전 3.3에 추가.

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

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)
Return a pointer to the canonical representation cast to UCS1, UCS2 or UCS4 integer types for direct character access. No checks are performed if the canonical representation has the correct character size; use `PyUnicode_KIND()` to select the right macro. Make sure `PyUnicode_READY()` has been called before accessing this.

버전 3.3에 추가.

PyUnicode_WCHAR_KIND
PyUnicode_1BYTE_KIND
PyUnicode_2BYTE_KIND
PyUnicode_4BYTE_KIND
Return values of the `PyUnicode_KIND()` macro.

버전 3.3에 추가.

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

버전 3.3에 추가.

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

버전 3.3에 추가.

void **PyUnicode_WRITE**(int *kind*, void **data*, Py_ssize_t *index*, *Py_UCS4 value*)
Write into a canonical representation *data* (as obtained with `PyUnicode_DATA()`). This macro does not do any sanity checks and is intended for usage in loops. The caller should cache the *kind* value and *data* pointer as obtained from other macro calls. *index* is the index in the string (starts at 0) and *value* is the new code point value which should be written to that location.

버전 3.3에 추가.

Py_UCS4 **PyUnicode_READ**(int *kind*, void **data*, Py_ssize_t *index*)
Read a code point from a canonical representation *data* (as obtained with `PyUnicode_DATA()`). No checks or ready calls are performed.

버전 3.3에 추가.

Py_UCS4 **PyUnicode_READ_CHAR**(*PyObject* *o, Py_ssize_t *index*)
Read a character from a Unicode object *o*, which must be in the "canonical" representation. This is less efficient than `PyUnicode_READ()` if you do multiple consecutive reads.

버전 3.3에 추가.

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

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

Deprecated since version 3.3, will be removed in version 4.0: Part of the old-style Unicode API, please migrate to using `PyUnicode_GET_LENGTH()`.

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

Deprecated since version 3.3, will be removed in version 4.0: Part of the old-style Unicode API, please migrate to using `PyUnicode_GET_LENGTH()`.

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

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

Deprecated since version 3.3, will be removed in version 4.0: Part of the old-style Unicode API, please migrate to using the `PyUnicode_nBYTE_DATA()` family of macros.

Unicode Character Properties

Unicode provides many different character properties. The most often needed ones are available through these macros which are mapped to C functions depending on the Python configuration.

int **Py_UNICODE_ISSPACE**(*Py_UNICODE ch*)
Return `1` or `0` depending on whether *ch* is a whitespace character.

int **Py_UNICODE_ISLOWER**(*Py_UNICODE ch*)
Return `1` or `0` depending on whether *ch* is a lowercase character.

int **Py_UNICODE_ISUPPER**(*Py_UNICODE ch*)
Return `1` or `0` depending on whether *ch* is an uppercase character.

int **Py_UNICODE_ISTITLE**(*Py_UNICODE ch*)
Return `1` or `0` depending on whether *ch* is a titlecase character.

int **Py_UNICODE_ISLINEBREAK**(*Py_UNICODE ch*)
Return `1` or `0` depending on whether *ch* is a linebreak character.

int **Py_UNICODE_ISDECIMAL**(*Py_UNICODE ch*)
Return `1` or `0` depending on whether *ch* is a decimal character.

int **Py_UNICODE_ISDIGIT**(*Py_UNICODE ch*)
Return `1` or `0` depending on whether *ch* is a digit character.

int **Py_UNICODE_ISNUMERIC**(*Py_UNICODE ch*)
Return `1` or `0` depending on whether *ch* is a numeric character.

int **Py_UNICODE_ISALPHA**(*Py_UNICODE ch*)
Return `1` or `0` depending on whether *ch* is an alphabetic character.

int **Py_UNICODE_ISALNUM**(*Py_UNICODE ch*)
Return `1` or `0` depending on whether *ch* is an alphanumeric character.

int **Py_UNICODE_ISPRINTABLE**(*Py_UNICODE ch*)
Return `1` or `0` depending on whether *ch* is a printable character. Nonprintable characters are those characters defined in the Unicode character database as "Other" or "Separator", excepting the ASCII space (0x20) which is considered printable. (Note that printable characters in this context are those which should not be escaped when `repr()` is invoked on a string. It has no bearing on the handling of strings written to `sys.stdout` or `sys.stderr`.)

These APIs can be used for fast direct character conversions:

Py_UNICODE **Py_UNICODE_TOLOWER**(*Py_UNICODE ch*)
Return the character *ch* converted to lower case.

버전 3.3부터 폐지: This function uses simple case mappings.

Py_UNICODE **Py_UNICODE_TOUPPER**(*Py_UNICODE ch*)
Return the character *ch* converted to upper case.

버전 3.3부터 폐지: This function uses simple case mappings.

Py_UNICODE **Py_UNICODE_TOTITLE**(*Py_UNICODE ch*)
Return the character *ch* converted to title case.

버전 3.3부터 폐지: This function uses simple case mappings.

int **Py_UNICODE_TODECIMAL**(*Py_UNICODE ch*)
Return the character *ch* converted to a decimal positive integer. Return `-1` if this is not possible. This macro does not raise exceptions.

int **Py_UNICODE_TODIGIT**(*Py_UNICODE ch*)
Return the character *ch* converted to a single digit integer. Return `-1` if this is not possible. This macro does not raise exceptions.

double **Py_UNICODE_TONUMERIC**(*Py_UNICODE ch*)
Return the character *ch* converted to a double. Return `-1.0` if this is not possible. This macro does not raise exceptions.

These APIs can be used to work with surrogates:

Py_UNICODE_IS_SURROGATE(ch)
Check if *ch* is a surrogate (`0xD800 <= ch <= 0xDFFF`).

Py_UNICODE_IS_HIGH_SURROGATE(ch)
Check if *ch* is a high surrogate (`0xD800 <= ch <= 0xDBFF`).

Py_UNICODE_IS_LOW_SURROGATE(ch)
Check if *ch* is a low surrogate (`0xDC00 <= ch <= 0xDFFF`).

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

Creating and accessing Unicode strings

To create Unicode objects and access their basic sequence properties, use these APIs:

*PyObject** **PyUnicode_New**(Py_ssize_t *size*, *Py_UCS4 maxchar*)
Return value: New reference. Create a new Unicode object. *maxchar* should be the true maximum code point to be placed in the string. As an approximation, it can be rounded up to the nearest value in the sequence 127, 255, 65535, 1114111.

This is the recommended way to allocate a new Unicode object. Objects created using this function are not resizable.

버전 3.3에 추가.

*PyObject** **PyUnicode_FromKindAndData**(int *kind*, const void **buffer*, Py_ssize_t *size*)
Return value: New reference. Create a new Unicode object with the given *kind* (possible values are *PyUnicode_1BYTE_KIND* etc., as returned by *PyUnicode_KIND()*). The *buffer* must point to an array of *size* units of 1, 2 or 4 bytes per character, as given by the kind.

버전 3.3에 추가.

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

If *u* is `NULL`, this function behaves like *PyUnicode_FromUnicode()* with the buffer set to `NULL`. This usage is deprecated in favor of *PyUnicode_New()*.

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

*PyObject** **PyUnicode_FromFormat**(const char **format*, ...)
Return value: New reference. Take a C `printf()`-style *format* string and a variable number of arguments, calculate the size of the resulting Python Unicode string and return a string with the values formatted into it. The variable arguments must be C types and must correspond exactly to the format characters in the *format* ASCII-encoded string. The following format characters are allowed:

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

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

참고: The width formatter unit is number of characters rather than bytes. The precision formatter unit is number of bytes for "%s" and "%V" (if the PyObject* argument is NULL), and a number of characters for "%A", "%U", "%S", "%R" and "%V" (if the PyObject* argument is not NULL).

버전 3.2에서 변경: Support for "%lld" and "%llu" added.

버전 3.3에서 변경: Support for "%li", "%lli" and "%zi" added.

버전 3.4에서 변경: Support width and precision formatter for "%s", "%A", "%U", "%V", "%S", "%R" added.

*PyObject** **PyUnicode_FromFormatV**(const char **format*, va_list *vargs*)
Return value: New reference. Identical to *PyUnicode_FromFormat()* except that it takes exactly two arguments.

*PyObject** **PyUnicode_FromEncodedObject**(*PyObject* **obj*, const char **encoding*, const char **errors*)
Return value: New reference. Decode an encoded object *obj* to a Unicode object.

bytes, bytearray and other *bytes-like objects* are decoded according to the given *encoding* and using the error handling defined by *errors*. Both can be NULL to have the interface use the default values (see *Built-in Codecs* for details).

All other objects, including Unicode objects, cause a TypeError to be set.

The API returns NULL if there was an error. The caller is responsible for decref'ing the returned objects.

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

Py_ssize_t **PyUnicode_GetLength**(*PyObject* *unicode)
Return the length of the Unicode object, in code points.

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*)
Copy characters from one Unicode object into another. This function performs character conversion when necessary and falls back to `memcpy()` if possible. Returns `-1` and sets an exception on error, otherwise returns the number of copied characters.

버전 3.3에 추가.

Py_ssize_t **PyUnicode_Fill**(*PyObject* *unicode, Py_ssize_t *start*, Py_ssize_t *length*, *Py_UCS4 fill_char*)
Fill a string with a character: write *fill_char* into `unicode[start:start+length]`.

Fail if *fill_char* is bigger than the string maximum character, or if the string has more than 1 reference.

Return the number of written character, or return `-1` and raise an exception on error.

버전 3.3에 추가.

int **PyUnicode_WriteChar**(*PyObject* *unicode, Py_ssize_t *index*, *Py_UCS4 character*)
Write a character to a string. The string must have been created through *PyUnicode_New()*. Since Unicode strings are supposed to be immutable, the string must not be shared, or have been hashed yet.

This function checks that *unicode* is a Unicode object, that the index is not out of bounds, and that the object can be modified safely (i.e. that it its reference count is one).

버전 3.3에 추가.

Py_UCS4 **PyUnicode_ReadChar**(*PyObject* *unicode, Py_ssize_t *index*)
Read a character from a string. This function checks that *unicode* is a Unicode object and the index is not out of bounds, in contrast to the macro version *PyUnicode_READ_CHAR()*.

버전 3.3에 추가.

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

버전 3.3에 추가.

*Py_UCS4** **PyUnicode_AsUCS4**(*PyObject* *u, *Py_UCS4* *buffer, Py_ssize_t *buflen*, int *copy_null*)
Copy the string *u* into a UCS4 buffer, including a null character, if *copy_null* is set. Returns `NULL` and sets an exception on error (in particular, a `SystemError` if *buflen* is smaller than the length of *u*). *buffer* is returned on success.

버전 3.3에 추가.

*Py_UCS4** **PyUnicode_AsUCS4Copy**(*PyObject* *u)
Copy the string *u* into a new UCS4 buffer that is allocated using *PyMem_Malloc()*. If this fails, `NULL` is returned with a `MemoryError` set. The returned buffer always has an extra null code point appended.

버전 3.3에 추가.

Deprecated Py_UNICODE APIs

Deprecated since version 3.3, will be removed in version 4.0.

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

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

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

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

Please migrate to using *PyUnicode_FromKindAndData()*, *PyUnicode_FromWideChar()* or *PyUnicode_New()*.

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

Please migrate to using *PyUnicode_AsUCS4()*, *PyUnicode_AsWideChar()*, *PyUnicode_ReadChar()* or similar new APIs.

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

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

버전 3.3에 추가.

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

버전 3.2에 추가.

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

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

Please migrate to using *PyUnicode_GetLength()*.

*PyObject** **PyUnicode_FromObject**(*PyObject* *obj)
Return value: New reference. Copy an instance of a Unicode subtype to a new true Unicode object if necessary. If *obj* is already a true Unicode object (not a subtype), return the reference with incremented refcount.

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

Locale Encoding

The current locale encoding can be used to decode text from the operating system.

*PyObject** **PyUnicode_DecodeLocaleAndSize**(const char *str, Py_ssize_t *len*, const char *errors*)
Return value: New reference. Decode a string from UTF-8 on Android, or from the current locale encoding on other platforms. The supported error handlers are `"strict"` and `"surrogateescape"` (**PEP 383**). The decoder uses `"strict"` error handler if *errors* is `NULL`. *str* must end with a null character but cannot contain embedded null characters.

Use *PyUnicode_DecodeFSDefaultAndSize()* to decode a string from `Py_FileSystemDefaultEncoding` (the locale encoding read at Python startup).

This function ignores the Python UTF-8 mode.

더 보기:

The *Py_DecodeLocale()* function.

버전 3.3에 추가.

버전 3.7에서 변경: The function now also uses the current locale encoding for the `surrogateescape` error handler, except on Android. Previously, *Py_DecodeLocale()* was used for the `surrogateescape`, and the current locale encoding was used for `strict`.

*PyObject** **PyUnicode_DecodeLocale**(const char *str, const char *errors*)
Return value: New reference. Similar to *PyUnicode_DecodeLocaleAndSize()*, but compute the string length using `strlen()`.

버전 3.3에 추가.

*PyObject** **PyUnicode_EncodeLocale**(*PyObject* *unicode*, const char *errors*)
Return value: New reference. Encode a Unicode object to UTF-8 on Android, or to the current locale encoding on other platforms. The supported error handlers are `"strict"` and `"surrogateescape"` (**PEP 383**). The encoder uses `"strict"` error handler if *errors* is `NULL`. Return a `bytes` object. *unicode* cannot contain embedded null characters.

Use *PyUnicode_EncodeFSDefault()* to encode a string to `Py_FileSystemDefaultEncoding` (the locale encoding read at Python startup).

This function ignores the Python UTF-8 mode.

더 보기:

The *Py_EncodeLocale()* function.

버전 3.3에 추가.

버전 3.7에서 변경: The function now also uses the current locale encoding for the `surrogateescape` error handler, except on Android. Previously, *Py_EncodeLocale()* was used for the `surrogateescape`, and the current locale encoding was used for `strict`.

File System Encoding

To encode and decode file names and other environment strings, `Py_FileSystemDefaultEncoding` should be used as the encoding, and `Py_FileSystemDefaultEncodeErrors` should be used as the error handler (**PEP 383** and **PEP 529**). To encode file names to `bytes` during argument parsing, the `"O&"` converter should be used, passing *PyUnicode_FSConverter()* as the conversion function:

int **PyUnicode_FSConverter**(*PyObject** *obj*, void* *result*)
ParseTuple converter: encode `str` objects – obtained directly or through the `os.PathLike` interface – to `bytes` using *PyUnicode_EncodeFSDefault()*; `bytes` objects are output as-is. *result* must be a *PyBytesObject** which must be released when it is no longer used.

버전 3.1에 추가.

버전 3.6에서 변경: Accepts a *path-like object*.

To decode file names to `str` during argument parsing, the `"O&"` converter should be used, passing *`PyUnicode_FSDecoder()`* as the conversion function:

int **`PyUnicode_FSDecoder`**(*PyObject** *obj*, void* *result*)
ParseTuple converter: decode `bytes` objects – obtained either directly or indirectly through the `os.PathLike` interface – to `str` using *`PyUnicode_DecodeFSDefaultAndSize()`*; `str` objects are output as-is. *result* must be a *`PyUnicodeObject*`* which must be released when it is no longer used.

버전 3.2에 추가.

버전 3.6에서 변경: Accepts a *path-like object*.

*PyObject** **`PyUnicode_DecodeFSDefaultAndSize`**(const char **s*, Py_ssize_t *size*)
Return value: New reference. Decode a string using `Py_FileSystemDefaultEncoding` and the `Py_FileSystemDefaultEncodeErrors` error handler.

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

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

더 보기:

The *`Py_DecodeLocale()`* function.

버전 3.6에서 변경: Use `Py_FileSystemDefaultEncodeErrors` error handler.

*PyObject** **`PyUnicode_DecodeFSDefault`**(const char **s*)
Return value: New reference. Decode a null-terminated string using `Py_FileSystemDefaultEncoding` and the `Py_FileSystemDefaultEncodeErrors` error handler.

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

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

버전 3.6에서 변경: Use `Py_FileSystemDefaultEncodeErrors` error handler.

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

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

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

더 보기:

The *`Py_EncodeLocale()`* function.

버전 3.2에 추가.

버전 3.6에서 변경: Use `Py_FileSystemDefaultEncodeErrors` error handler.

wchar_t Support

`wchar_t` support for platforms which support it:

*PyObject** **PyUnicode_FromWideChar** (const wchar_t *w, Py_ssize_t *size*)
Return value: New reference. Create a Unicode object from the `wchar_t` buffer *w* of the given *size*. Passing `-1` as the *size* indicates that the function must itself compute the length, using wcslen. Return `NULL` on failure.

Py_ssize_t **PyUnicode_AsWideChar** (*PyObject* *unicode*, wchar_t *w, Py_ssize_t *size*)
Copy the Unicode object contents into the `wchar_t` buffer *w*. At most *size* `wchar_t` characters are copied (excluding a possibly trailing null termination character). Return the number of `wchar_t` characters copied or `-1` in case of an error. Note that the resulting `wchar_t*` string may or may not be null-terminated. It is the responsibility of the caller to make sure that the `wchar_t*` string is null-terminated in case this is required by the application. Also, note that the `wchar_t*` string might contain null characters, which would cause the string to be truncated when used with most C functions.

wchar_t* **PyUnicode_AsWideCharString** (*PyObject* *unicode*, Py_ssize_t *size*)
Convert the Unicode object to a wide character string. The output string always ends with a null character. If *size* is not `NULL`, write the number of wide characters (excluding the trailing null termination character) into *size*. Note that the resulting `wchar_t` string might contain null characters, which would cause the string to be truncated when used with most C functions. If *size* is `NULL` and the `wchar_t*` string contains null characters a `ValueError` is raised.

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

버전 3.2에 추가.

버전 3.7에서 변경: Raises a `ValueError` if *size* is `NULL` and the `wchar_t*` string contains null characters.

Built-in Codecs

Python provides a set of built-in codecs which are written in C for speed. All of these codecs are directly usable via the following functions.

Many of the following APIs take two arguments encoding and errors, and they have the same semantics as the ones of the built-in `str()` string object constructor.

Setting encoding to `NULL` causes the default encoding to be used which is 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).

Error handling is set by errors which may also be set to `NULL` meaning to use the default handling defined for the codec. Default error handling for all built-in codecs is "strict" (`ValueError` is raised).

The codecs all use a similar interface. Only deviation from the following generic ones are documented for simplicity.

Generic Codecs

These are the generic codec APIs:

*PyObject** **PyUnicode_Decode** (const char *s, Py_ssize_t *size*, const char *encoding*, const char *errors*)
Return value: New reference. Create a Unicode object by decoding *size* bytes of the encoded string *s*. *encoding* and *errors* have the same meaning as the parameters of the same name in the `str()` built-in function. The codec to be used is looked up using the Python codec registry. Return `NULL` if an exception was raised by the codec.

*PyObject** **PyUnicode_AsEncodedString**(*PyObject* *unicode*, const char *encoding*, const char *errors*)

Return value: New reference. Encode a Unicode object and return the result as Python bytes object. *encoding* and *errors* have the same meaning as the parameters of the same name in the Unicode `encode()` method. The codec to be used is looked up using the Python codec registry. Return `NULL` if an exception was raised by the codec.

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

Return value: New reference. Encode the *Py_UNICODE* buffer *s* of the given *size* and return a Python bytes object. *encoding* and *errors* have the same meaning as the parameters of the same name in the Unicode `encode()` method. The codec to be used is looked up using the Python codec registry. Return `NULL` if an exception was raised by the codec.

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

UTF-8 Codecs

These are the UTF-8 codec APIs:

*PyObject** **PyUnicode_DecodeUTF8**(const char *s, Py_ssize_t *size*, const char *errors*)

Return value: New reference. Create a Unicode object by decoding *size* bytes of the UTF-8 encoded string *s*. Return `NULL` if an exception was raised by the codec.

*PyObject** **PyUnicode_DecodeUTF8Stateful**(const char *s, Py_ssize_t *size*, const char *errors*, Py_ssize_t *consumed*)

Return value: New reference. If *consumed* is `NULL`, behave like *PyUnicode_DecodeUTF8()*. If *consumed* is not `NULL`, trailing incomplete UTF-8 byte sequences will not be treated as an error. Those bytes will not be decoded and the number of bytes that have been decoded will be stored in *consumed*.

*PyObject** **PyUnicode_AsUTF8String**(*PyObject* *unicode*)

Return value: New reference. Encode a Unicode object using UTF-8 and return the result as Python bytes object. Error handling is "strict". Return `NULL` if an exception was raised by the codec.

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

Return a pointer to the UTF-8 encoding of the Unicode object, and store the size of the encoded representation (in bytes) in *size*. The *size* argument can be `NULL`; in this case no size will be stored. The returned buffer always has an extra null byte appended (not included in *size*), regardless of whether there are any other null code points.

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

This caches the UTF-8 representation of the string in the Unicode object, and subsequent calls will return a pointer to the same buffer. The caller is not responsible for deallocating the buffer.

버전 3.3에 추가.

버전 3.7에서 변경: The return type is now `const char *` rather of `char *`.

const char* **PyUnicode_AsUTF8**(*PyObject* *unicode*)

As *PyUnicode_AsUTF8AndSize()*, but does not store the size.

버전 3.3에 추가.

버전 3.7에서 변경: The return type is now `const char *` rather of `char *`.

*PyObject** **PyUnicode_EncodeUTF8**(const *Py_UNICODE* *s*, Py_ssize_t *size*, const char *errors*)

Return value: New reference. Encode the *Py_UNICODE* buffer *s* of the given *size* using UTF-8 and return a Python bytes object. Return `NULL` if an exception was raised by the codec.

Deprecated since version 3.3, will be removed in version 4.0: Part of the old-style *Py_UNICODE* API; please migrate to using *PyUnicode_AsUTF8String()*, *PyUnicode_AsUTF8AndSize()* or *PyUnicode_AsEncodedString()*.

UTF-32 Codecs

These are the UTF-32 codec APIs:

*PyObject** **PyUnicode_DecodeUTF32**(const char *s, Py_ssize_t *size*, const char *errors, int *byteorder)

Return value: New reference. Decode *size* bytes from a UTF-32 encoded buffer string and return the corresponding Unicode object. *errors* (if non-NULL) defines the error handling. It defaults to "strict".

If *byteorder* is non-NULL, the decoder starts decoding using the given byte order:

```
*byteorder == -1: little endian
*byteorder == 0:  native order
*byteorder == 1:  big endian
```

If *byteorder is zero, and the first four bytes of the input data are a byte order mark (BOM), the decoder switches to this byte order and the BOM is not copied into the resulting Unicode string. If *byteorder is -1 or 1, any byte order mark is copied to the output.

After completion, *byteorder is set to the current byte order at the end of input data.

If *byteorder* is NULL, the codec starts in native order mode.

Return NULL if an exception was raised by the codec.

*PyObject** **PyUnicode_DecodeUTF32Stateful**(const char *s, Py_ssize_t *size*, const char *errors, int *byteorder, Py_ssize_t *consumed)

Return value: New reference. If *consumed* is NULL, behave like *PyUnicode_DecodeUTF32()*. If *consumed* is not NULL, *PyUnicode_DecodeUTF32Stateful()* will not treat trailing incomplete UTF-32 byte sequences (such as a number of bytes not divisible by four) as an error. Those bytes will not be decoded and the number of bytes that have been decoded will be stored in *consumed*.

*PyObject** **PyUnicode_AsUTF32String**(*PyObject* *unicode)

Return value: New reference. Return a Python byte string using the UTF-32 encoding in native byte order. The string always starts with a BOM mark. Error handling is "strict". Return NULL if an exception was raised by the codec.

*PyObject** **PyUnicode_EncodeUTF32**(const *Py_UNICODE* *s, Py_ssize_t *size*, const char *errors, int *byteorder*)

Return value: New reference. Return a Python bytes object holding the UTF-32 encoded value of the Unicode data in *s*. Output is written according to the following byte order:

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

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

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

Return NULL if an exception was raised by the codec.

Deprecated since version 3.3, will be removed in version 4.0: Part of the old-style *Py_UNICODE* API; please migrate to using *PyUnicode_AsUTF32String()* or *PyUnicode_AsEncodedString()*.

UTF-16 Codecs

These are the UTF-16 codec APIs:

*PyObject** **PyUnicode_DecodeUTF16**(const char *s, Py_ssize_t *size*, const char *errors, int *byteorder)

Return value: New reference. Decode *size* bytes from a UTF-16 encoded buffer string and return the corresponding Unicode object. *errors* (if non-NULL) defines the error handling. It defaults to "strict".

If *byteorder* is non-NULL, the decoder starts decoding using the given byte order:

```
*byteorder == -1: little endian
*byteorder == 0:  native order
*byteorder == 1:  big endian
```

If *byteorder is zero, and the first two bytes of the input data are a byte order mark (BOM), the decoder switches to this byte order and the BOM is not copied into the resulting Unicode string. If *byteorder is -1 or 1, any byte order mark is copied to the output (where it will result in either a \ufeff or a \ufffe character).

After completion, *byteorder is set to the current byte order at the end of input data.

If *byteorder* is NULL, the codec starts in native order mode.

Return NULL if an exception was raised by the codec.

*PyObject** **PyUnicode_DecodeUTF16Stateful**(const char *s, Py_ssize_t *size*, const char *errors, int *byteorder, Py_ssize_t *consumed)

Return value: New reference. If *consumed* is NULL, behave like *PyUnicode_DecodeUTF16()*. If *consumed* is not NULL, *PyUnicode_DecodeUTF16Stateful()* will not treat trailing incomplete UTF-16 byte sequences (such as an odd number of bytes or a split surrogate pair) as an error. Those bytes will not be decoded and the number of bytes that have been decoded will be stored in *consumed*.

*PyObject** **PyUnicode_AsUTF16String**(*PyObject* *unicode)

Return value: New reference. Return a Python byte string using the UTF-16 encoding in native byte order. The string always starts with a BOM mark. Error handling is "strict". Return NULL if an exception was raised by the codec.

*PyObject** **PyUnicode_EncodeUTF16**(const *Py_UNICODE* *s, Py_ssize_t *size*, const char *errors, int *byteorder*)

Return value: New reference. Return a Python bytes object holding the UTF-16 encoded value of the Unicode data in *s*. Output is written according to the following byte order:

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

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

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

Return NULL if an exception was raised by the codec.

Deprecated since version 3.3, will be removed in version 4.0: Part of the old-style *Py_UNICODE* API; please migrate to using *PyUnicode_AsUTF16String()* or *PyUnicode_AsEncodedString()*.

UTF-7 Codecs

These are the UTF-7 codec APIs:

*PyObject** **PyUnicode_DecodeUTF7**(const char *s, Py_ssize_t *size*, const char *errors*)
Return value: New reference. Create a Unicode object by decoding *size* bytes of the UTF-7 encoded string *s*. Return `NULL` if an exception was raised by the codec.

*PyObject** **PyUnicode_DecodeUTF7Stateful**(const char *s, Py_ssize_t *size*, const char *errors*, Py_ssize_t *consumed*)
Return value: New reference. If *consumed* is `NULL`, behave like `PyUnicode_DecodeUTF7()`. If *consumed* is not `NULL`, trailing incomplete UTF-7 base-64 sections will not be treated as an error. Those bytes will not be decoded and the number of bytes that have been decoded will be stored in *consumed*.

*PyObject** **PyUnicode_EncodeUTF7**(const *Py_UNICODE* *s, Py_ssize_t *size*, int *base64SetO*, int *base64WhiteSpace*, const char *errors*)
Return value: New reference. Encode the `Py_UNICODE` buffer of the given size using UTF-7 and return a Python bytes object. Return `NULL` if an exception was raised by the codec.

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

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

Unicode-Escape Codecs

These are the "Unicode Escape" codec APIs:

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

*PyObject** **PyUnicode_AsUnicodeEscapeString**(*PyObject* *unicode*)
Return value: New reference. Encode a Unicode object using Unicode-Escape and return the result as a bytes object. Error handling is "strict". Return `NULL` if an exception was raised by the codec.

*PyObject** **PyUnicode_EncodeUnicodeEscape**(const *Py_UNICODE* *s, Py_ssize_t *size*)
Return value: New reference. Encode the `Py_UNICODE` buffer of the given *size* using Unicode-Escape and return a bytes object. Return `NULL` if an exception was raised by the codec.

Deprecated since version 3.3, will be removed in version 4.0: Part of the old-style `Py_UNICODE` API; please migrate to using `PyUnicode_AsUnicodeEscapeString()`.

Raw-Unicode-Escape Codecs

These are the "Raw Unicode Escape" codec APIs:

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

*PyObject** **PyUnicode_AsRawUnicodeEscapeString**(*PyObject* *unicode*)
Return value: New reference. Encode a Unicode object using Raw-Unicode-Escape and return the result as a bytes object. Error handling is "strict". Return `NULL` if an exception was raised by the codec.

*PyObject** **PyUnicode_EncodeRawUnicodeEscape**(const *Py_UNICODE* *s, Py_ssize_t *size*)
Return value: New reference. Encode the `Py_UNICODE` buffer of the given *size* using Raw-Unicode-Escape and return a bytes object. Return `NULL` if an exception was raised by the codec.

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

Latin-1 Codecs

These are the Latin-1 codec APIs: Latin-1 corresponds to the first 256 Unicode ordinals and only these are accepted by the codecs during encoding.

*PyObject** **PyUnicode_DecodeLatin1**(const char **s*, Py_ssize_t *size*, const char **errors*)
Return value: New reference. Create a Unicode object by decoding *size* bytes of the Latin-1 encoded string *s*. Return `NULL` if an exception was raised by the codec.

*PyObject** **PyUnicode_AsLatin1String**(*PyObject* **unicode*)
Return value: New reference. Encode a Unicode object using Latin-1 and return the result as Python bytes object. Error handling is "strict". Return `NULL` if an exception was raised by the codec.

*PyObject** **PyUnicode_EncodeLatin1**(const *Py_UNICODE* **s*, Py_ssize_t *size*, const char **errors*)
Return value: New reference. Encode the *Py_UNICODE* buffer of the given *size* using Latin-1 and return a Python bytes object. Return `NULL` if an exception was raised by the codec.

Deprecated since version 3.3, will be removed in version 4.0: Part of the old-style *Py_UNICODE* API; please migrate to using *PyUnicode_AsLatin1String()* or *PyUnicode_AsEncodedString()*.

ASCII Codecs

These are the ASCII codec APIs. Only 7-bit ASCII data is accepted. All other codes generate errors.

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

*PyObject** **PyUnicode_AsASCIIString**(*PyObject* **unicode*)
Return value: New reference. Encode a Unicode object using ASCII and return the result as Python bytes object. Error handling is "strict". Return `NULL` if an exception was raised by the codec.

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

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

Character Map Codecs

This codec is special in that it can be used to implement many different codecs (and this is in fact what was done to obtain most of the standard codecs included in the `encodings` package). The codec uses mapping to encode and decode characters. The mapping objects provided must support the `__getitem__()` mapping interface; dictionaries and sequences work well.

These are the mapping codec APIs:

*PyObject** **PyUnicode_DecodeCharmap**(const char **data*, Py_ssize_t *size*, *PyObject* **mapping*, const char **errors*)
Return value: New reference. Create a Unicode object by decoding *size* bytes of the encoded string *s* using the given *mapping* object. Return `NULL` if an exception was raised by the codec.

If *mapping* is `NULL`, Latin-1 decoding will be applied. Else *mapping* must map bytes ordinals (integers in the range from 0 to 255) to Unicode strings, integers (which are then interpreted as Unicode ordinals) or `None`. Unmapped data bytes – ones which cause a `LookupError`, as well as ones which get mapped to `None`, `0xFFFE` or `'\ufffe'`, are treated as undefined mappings and cause an error.

*PyObject** **PyUnicode_AsCharmapString**(*PyObject* *unicode*, *PyObject* *mapping*)
Return value: New reference. Encode a Unicode object using the given *mapping* object and return the result as a bytes object. Error handling is "strict". Return `NULL` if an exception was raised by the codec.

The *mapping* object must map Unicode ordinal integers to bytes objects, integers in the range from 0 to 255 or `None`. Unmapped character ordinals (ones which cause a `LookupError`) as well as mapped to `None` are treated as "undefined mapping" and cause an error.

*PyObject** **PyUnicode_EncodeCharmap**(const *Py_UNICODE* *s, Py_ssize_t *size*, *PyObject* *mapping*, const char *errors*)
Return value: New reference. Encode the *Py_UNICODE* buffer of the given *size* using the given *mapping* object and return the result as a bytes object. Return `NULL` if an exception was raised by the codec.

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

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

*PyObject** **PyUnicode_Translate**(*PyObject* *unicode*, *PyObject* *mapping*, const char *errors*)
Return value: New reference. Translate a Unicode object using the given *mapping* object and return the resulting Unicode object. Return `NULL` if an exception was raised by the codec.

The *mapping* object must map Unicode ordinal integers to Unicode strings, integers (which are then interpreted as Unicode ordinals) or `None` (causing deletion of the character). Unmapped character ordinals (ones which cause a `LookupError`) are left untouched and are copied as-is.

*PyObject** **PyUnicode_TranslateCharmap**(const *Py_UNICODE* *s, Py_ssize_t *size*, *PyObject* *mapping*, const char *errors*)
Return value: New reference. Translate a *Py_UNICODE* buffer of the given *size* by applying a character *mapping* table to it and return the resulting Unicode object. Return `NULL` when an exception was raised by the codec.

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

MBCS codecs for Windows

These are the MBCS codec APIs. They are currently only available on Windows and use the Win32 MBCS converters to implement the conversions. Note that MBCS (or DBCS) is a class of encodings, not just one. The target encoding is defined by the user settings on the machine running the codec.

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

*PyObject** **PyUnicode_DecodeMBCSStateful**(const char *s, Py_ssize_t *size*, const char *errors*, Py_ssize_t *consumed*)
Return value: New reference. If *consumed* is `NULL`, behave like *PyUnicode_DecodeMBCS()*. If *consumed* is not `NULL`, *PyUnicode_DecodeMBCSStateful()* will not decode trailing lead byte and the number of bytes that have been decoded will be stored in *consumed*.

*PyObject** **PyUnicode_AsMBCSString**(*PyObject* *unicode*)
Return value: New reference. Encode a Unicode object using MBCS and return the result as Python bytes object. Error handling is "strict". Return `NULL` if an exception was raised by the codec.

*PyObject** **PyUnicode_EncodeCodePage**(int *code_page*, *PyObject* *unicode*, const char *errors*)
Return value: New reference. Encode the Unicode object using the specified code page and return a Python bytes object. Return `NULL` if an exception was raised by the codec. Use `CP_ACP` code page to get the MBCS encoder.

버전 3.3에 추가.

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

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

Methods & Slots

Methods and Slot Functions

The following APIs are capable of handling Unicode objects and strings on input (we refer to them as strings in the descriptions) and return Unicode objects or integers as appropriate.

They all return `NULL` or `-1` if an exception occurs.

*PyObject** **PyUnicode_Concat**(*PyObject* **left*, *PyObject* **right*)
Return value: New reference. Concat two strings giving a new Unicode string.

*PyObject** **PyUnicode_Split**(*PyObject* **s*, *PyObject* **sep*, Py_ssize_t *maxsplit*)
Return value: New reference. Split a string giving a list of Unicode strings. If *sep* is `NULL`, splitting will be done at all whitespace substrings. Otherwise, splits occur at the given separator. At most *maxsplit* splits will be done. If negative, no limit is set. Separators are not included in the resulting list.

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

*PyObject** **PyUnicode_Translate**(*PyObject* **str*, *PyObject* **table*, const char **errors*)
Translate a string by applying a character mapping table to it and return the resulting Unicode object.

The mapping table must map Unicode ordinal integers to Unicode ordinal integers or `None` (causing deletion of the character).

Mapping tables need only provide the `__getitem__()` interface; dictionaries and sequences work well. Unmapped character ordinals (ones which cause a `LookupError`) are left untouched and are copied as-is.

errors has the usual meaning for codecs. It may be `NULL` which indicates to use the default error handling.

*PyObject** **PyUnicode_Join**(*PyObject* **separator*, *PyObject* **seq*)
Return value: New reference. Join a sequence of strings using the given *separator* and return the resulting Unicode string.

Py_ssize_t **PyUnicode_Tailmatch**(*PyObject* **str*, *PyObject* **substr*, Py_ssize_t *start*, Py_ssize_t *end*, int *direction*)
Return `1` if *substr* matches `str[start:end]` at the given tail end (*direction* == `-1` means to do a prefix match, *direction* == `1` a suffix match), `0` otherwise. Return `-1` if an error occurred.

Py_ssize_t **PyUnicode_Find**(*PyObject* **str*, *PyObject* **substr*, Py_ssize_t *start*, Py_ssize_t *end*, int *direction*)
Return the first position of *substr* in `str[start:end]` using the given *direction* (*direction* == `1` means to do a forward search, *direction* == `-1` a backward search). The return value is the index of the first match; a value of `-1` indicates that no match was found, and `-2` indicates that an error occurred and an exception has been set.

Py_ssize_t **PyUnicode_FindChar**(*PyObject* **str*, *Py_UCS4* *ch*, Py_ssize_t *start*, Py_ssize_t *end*, int *direction*)
Return the first position of the character *ch* in `str[start:end]` using the given *direction* (*direction* == `1` means to do a forward search, *direction* == `-1` a backward search). The return value is the index of the first match; a value of `-1` indicates that no match was found, and `-2` indicates that an error occurred and an exception has been set.

버전 3.3에 추가.

버전 3.7에서 변경: *start* and *end* are now adjusted to behave like `str[start:end]`.

Py_ssize_t **PyUnicode_Count**(*PyObject* *str, *PyObject* *substr, Py_ssize_t *start*, Py_ssize_t *end*)
Return the number of non-overlapping occurrences of *substr* in `str[start:end]`. Return `-1` if an error occurred.

*PyObject** **PyUnicode_Replace**(*PyObject* *str, *PyObject* *substr, *PyObject* *replstr, Py_ssize_t *maxcount*)
Return value: New reference. Replace at most *maxcount* occurrences of *substr* in *str* with *replstr* and return the resulting Unicode object. *maxcount* == `-1` means replace all occurrences.

int **PyUnicode_Compare**(*PyObject* *left, *PyObject* *right)
Compare two strings and return `-1`, `0`, `1` for less than, equal, and greater than, respectively.

This function returns `-1` upon failure, so one should call *PyErr_Occurred()* to check for errors.

int **PyUnicode_CompareWithASCIIString**(*PyObject* *uni, const char *string)
Compare a Unicode object, *uni*, with *string* and return `-1`, `0`, `1` for less than, equal, and greater than, respectively. It is best to pass only ASCII-encoded strings, but the function interprets the input string as ISO-8859-1 if it contains non-ASCII characters.

This function does not raise exceptions.

*PyObject** **PyUnicode_RichCompare**(*PyObject* *left, *PyObject* *right, int *op*)
Return value: New reference. Rich compare two Unicode strings and return one of the following:

- `NULL` in case an exception was raised
- `Py_True` or `Py_False` for successful comparisons
- `Py_NotImplemented` in case the type combination is unknown

Possible values for *op* are `Py_GT`, `Py_GE`, `Py_EQ`, `Py_NE`, `Py_LT`, and `Py_LE`.

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

int **PyUnicode_Contains**(*PyObject* *container, *PyObject* *element)
Check whether *element* is contained in *container* and return true or false accordingly.

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

void **PyUnicode_InternInPlace**(*PyObject* **string)
Intern the argument **string* in place. The argument must be the address of a pointer variable pointing to a Python Unicode string object. If there is an existing interned string that is the same as **string*, it sets **string* to it (decrementing the reference count of the old string object and incrementing the reference count of the interned string object), otherwise it leaves **string* alone and interns it (incrementing its reference count). (Clarification: even though there is a lot of talk about reference counts, think of this function as reference-count-neutral; you own the object after the call if and only if you owned it before the call.)

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

8.3.4 튜플 객체

PyTupleObject
이 *PyObject*의 서브 형은 파이썬 튜플 객체를 나타냅니다.

PyTypeObject **PyTuple_Type**
이 *PyTypeObject* 인스턴스는 파이썬 튜플 형을 나타냅니다. 파이썬 계층의 `tuple`과 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. Return a new tuple object of size *len*, or `NULL` on failure.

*PyObject** **PyTuple_Pack**(Py_ssize_t *n*, ...)
Return value: New reference. Return a new tuple object of size *n*, or `NULL` on failure. The tuple values are initialized to the subsequent *n* C arguments pointing to Python objects. `PyTuple_Pack(2, a, b)` is equivalent to `Py_BuildValue("(OO)", a, b)`.

Py_ssize_t **PyTuple_Size**(*PyObject* *p)
튜플 객체에 대한 포인터를 받아서, 해당 튜플의 크기를 반환합니다.

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

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

*PyObject** **PyTuple_GET_ITEM**(*PyObject* *p, Py_ssize_t *pos*)
Return value: Borrowed reference. `PyTuple_GetItem()`와 비슷하지만, 인자를 확인하지 않습니다.

*PyObject** **PyTuple_GetSlice**(*PyObject* *p, Py_ssize_t *low*, Py_ssize_t *high*)
Return value: New reference. Return the slice of the tuple pointed to by *p* between *low* and *high*, or `NULL` on failure. This is the equivalent of the Python expression `p[low:high]`. Indexing from the end of the list is not supported.

int **PyTuple_SetItem**(*PyObject* *p, Py_ssize_t *pos*, *PyObject* *o)
Insert a reference to object *o* at position *pos* of the tuple pointed to by *p*. Return `0` on success. If *pos* is out of bounds, return `-1` and set an `IndexError` exception.

참고: This function "steals" a reference to *o* and discards a reference to an item already in the tuple at the affected position.

---

void **PyTuple_SET_ITEM**(*PyObject* *p, Py_ssize_t *pos*, *PyObject* *o)
`PyTuple_SetItem()`과 비슷하지만, 에러 검사는 하지 않으며 새로운 튜플을 채울 때*만* 사용해야 합니다.

참고: This macro "steals" a reference to *o*, and, unlike `PyTuple_SetItem()`, does *not* discard a reference to any item that is being replaced; any reference in the tuple at position *pos* will be leaked.

---

int **_PyTuple_Resize**(*PyObject* **p, Py_ssize_t *newsize*)
Can be used to resize a tuple. *newsize* will be the new length of the tuple. Because tuples are *supposed* to be immutable, this should only be used if there is only one reference to the object. Do *not* use this if the tuple may already be known to some other part of the code. The tuple will always grow or shrink at the end. Think of this as destroying the old tuple and creating a new one, only more efficiently. Returns `0` on success. Client code should never assume that the resulting value of `*p` will be the same as before calling this function. If the object referenced by `*p` is replaced, the original `*p` is destroyed. On failure, returns `-1` and sets `*p` to `NULL`, and raises `MemoryError` or `SystemError`.

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.

*PyTypeObject** **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**(*PyTypeObject *type*, *PyStructSequence_Desc *desc*)
*desc*로 구조체 시퀀스 형 *type*을 재자리에서 초기화합니다.

int **PyStructSequence_InitType2**(*PyTypeObject *type*, *PyStructSequence_Desc *desc*)
`PyStructSequence_InitType`와 같지만, 성공하면 `0`을, 실패하면 `-1`을 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 *`	pointer to docstring for the type or `NULL` to omit
`fields`	`PyStructSequence_Field *`	pointer to `NULL`-terminated array with field names of the new type
`n_in_sequence`	`int`	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 *`	name for the field or `NULL` to end the list of named fields, set to *PyStructSequence_UnnamedField* to leave unnamed
`doc`	`const char *`	field docstring or `NULL` to omit

char* **PyStructSequence_UnnamedField**
이름 없는 상태로 남겨두기 위한 필드 이름의 특수 값.

*PyObject** **PyStructSequence_New**(*PyTypeObject *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. Return a new list of length *len* on success, or `NULL` on failure.

> **참고:** *len*이 0보다 크면, 반환된 리스트 객체의 항목은 `NULL`로 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. Return the object at position *index* in the list pointed to by *list*. The position must be non-negative; indexing from the end of the list is not supported. If *index* is out of bounds (<0 or >=len(list)), return `NULL` and set an `IndexError` exception.

*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)
Set the item at index *index* in list to *item*. Return `0` on success. If *index* is out of bounds, return `-1` and set an `IndexError` exception.

> **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. Return a list of the objects in *list* containing the objects *between low* and *high*. Return `NULL` and set an exception if unsuccessful. Analogous to `list[low:high]`. Indexing from the end of the list is not supported.

int **PyList_SetSlice**(*PyObject* *list, Py_ssize_t low, Py_ssize_t high, *PyObject* *itemlist)
Set the slice of *list* between *low* and *high* to the contents of *itemlist*. Analogous to `list[low:high] = itemlist`. The *itemlist* may be `NULL`, indicating the assignment of an empty list (slice deletion). Return `0` on success, `-1` on failure. Indexing from the end of the list is not supported.

int **PyList_Sort**(*PyObject* *list)
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. Return a new empty dictionary, or `NULL` on failure.

*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 `1`을 반환하고, 그렇지 않으면 `0`을 반환합니다. 에러면 `-1`을 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 *value*를 *key* 키로 삽입합니다. *key*는 해시 가능해야 합니다. 그렇지 않으면 `TypeError`가 발생합니다. 성공하면 `0`을, 실패하면 `-1`을 반환합니다.

int **PyDict_SetItemString**(*PyObject* *p, const char *key, *PyObject* *val)
*key*를 키로 사용하여 딕셔너리 *p*에 *value*를 삽입합니다. *key*는 `const char*`여야 합니다. 키 객체는 `PyUnicode_FromString(key)`를 사용하여 만듭니다. 성공하면 `0`을, 실패하면 `-1`을 반환합니다.

int **PyDict_DelItem**(*PyObject* *p, *PyObject* *key)
딕셔너리 *p*에서 키가 *key*인 항목을 제거합니다. *key*는 해시 가능해야 합니다. 그렇지 않으면 `TypeError`가 발생합니다. 성공하면 `0`을, 실패하면 `-1`을 반환합니다.

int **PyDict_DelItemString**(*PyObject* *p, const char *key)
딕셔너리 *p*에서 문자열 *key*로 지정된 키의 항목을 제거합니다. 성공하면 `0`을, 실패하면 `-1`을 반환합니다.

*PyObject** **PyDict_GetItem**(*PyObject* *p, *PyObject* *key)
Return value: Borrowed reference. Return the object from dictionary *p* which has a key *key*. Return `NULL` if the key *key* is not present, but *without* setting an exception.

`__hash__()`와 `__eq__()` 메서드를 호출하는 동안 발생하는 예외는 억제됩니다. 에러 보고를 얻으려면 대신 *PyDict_GetItemWithError()*를 사용하십시오.

*PyObject** **PyDict_GetItemWithError**(*PyObject* *p, *PyObject* *key)
Return value: Borrowed reference. Variant of *PyDict_GetItem()* that does not suppress exceptions. Return `NULL` **with** an exception set if an exception occurred. Return `NULL` **without** an exception set if the key wasn't present.

*PyObject** **PyDict_GetItemString**(*PyObject* *p, const char *key)
Return value: Borrowed reference. 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. 에러 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)
Iterate over all key-value pairs in the dictionary *p*. The `Py_ssize_t` referred to by *ppos* must be initialized to `0` prior to the first call to this function to start the iteration; the function returns true for each pair in the dictionary, and false once all pairs have been reported. The parameters *pkey* and *pvalue* should either point to *PyObject** variables that will be filled in with each key and value, respectively, or may be `NULL`. Any

references returned through them are borrowed. *ppos* should not be altered during iteration. Its value represents offsets within the internal dictionary structure, and since the structure is sparse, the offsets are not consecutive.

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 `0`을 반환하고, 예외가 발생하면 `-1`을 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 `0`을 반환하고, 예외가 발생하면 `-1`을 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 `0`을 반환하고, 예외가 발생하면 `-1`을 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 set과 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 set과 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. Return a new set containing objects returned by the *iterable*. The *iterable* may be NULL to create a new empty set. Return the new set on success or NULL on failure. Raise TypeError if *iterable* is not actually iterable. The constructor is also useful for copying a set (c=set(s)).

*PyObject** **PyFrozenSet_New**(*PyObject* *iterable)
Return value: New reference. Return a new frozenset containing objects returned by the *iterable*. The *iterable* may be NULL to create a new empty frozenset. Return the new set on success or NULL on failure. Raise TypeError if *iterable* is not actually iterable.

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_SystemError를 발생시킵니다.

Py_ssize_t **PySet_GET_SIZE**(*PyObject* *anyset)
에러 검사 없는 *PySet_Size()*의 매크로 형식.

int **PySet_Contains**(*PyObject* *anyset, *PyObject* *key)
발견되면 1을, 발견되지 않으면 0을, 에러가 발생하면 -1을 반환합니다. 파이썬 __contains__() 메서드와는 달리, 이 함수는 해시 불가능한 집합을 임시 frozenset으로 자동 변환하지 않습니다. *key*

가 해시 불가능하면, TypeError를 발생시킵니다. *anyset*이 set, frozenset 또는 서브 형의 인스턴스가 아니면 PyExc_SystemError를 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, TypeError를 발생시킵니다. 성장할 공간이 없다면 MemoryError를 발생시킵니다. *set*이 set 이나 그 서브 형의 인스턴스가 아니면 SystemError를 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, 에러가 발생하면 -1을 반환합니다. 발견할 수 없는 키에 대해 KeyError를 발생시키지 않습니다. *key*가 해시 불가능하면 TypeError를 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_SystemError를 발생시킵니다.

*PyObject** **PySet_Pop**(*PyObject* *set)
Return value: New reference. Return a new reference to an arbitrary object in the *set*, and removes the object from the *set*. Return NULL on failure. Raise KeyError if the set is empty. Raise a SystemError if *set* is not an instance of set or its subtype.

int **PySet_Clear**(*PyObject* *set)
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.

PyTypeObject **PyFunction_Type**
이것은 *PyTypeObject*의 인스턴스이며 파이썬 함수 형을 나타냅니다. 파이썬 프로그래머에게 types.FunctionType으로 노출됩니다.

int **PyFunction_Check**(*PyObject* *o)
Return true if *o* is a function object (has type *PyFunction_Type*). The parameter must not be NULL.

*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.

The function's docstring and name are retrieved from the code object. *__module__* is retrieved from *globals*. The argument defaults, annotations and closure are set to NULL. *__qualname__* is set to the same value as the function's name.

*PyObject** **PyFunction_NewWithQualName**(*PyObject* *code, *PyObject* *globals, *PyObject* *qualname)
Return value: New reference. As *PyFunction_New()*, but also allows setting the function object's __qualname__ attribute. *qualname* should be a unicode object or NULL; if NULL, the __qualname__ attribute is set to the same value as its __name__ attribute.

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. Return the argument default values of the function object *op*. This can be a tuple of arguments or `NULL`.

int **PyFunction_SetDefaults**(*PyObject *op*, *PyObject *defaults*)
Set the argument default values for the function object *op*. *defaults* must be `Py_None` or a tuple.

실패하면 `SystemError`를 발생시키고 `-1`을 반환합니다.

*PyObject** **PyFunction_GetClosure**(*PyObject *op*)
Return value: Borrowed reference. Return the closure associated with the function object *op*. This can be `NULL` or a tuple of cell objects.

int **PyFunction_SetClosure**(*PyObject *op*, *PyObject *closure*)
Set the closure associated with the function object *op*. *closure* must be `Py_None` or a tuple of cell objects.

실패하면 `SystemError`를 발생시키고 `-1`을 반환합니다.

PyObject ***PyFunction_GetAnnotations**(*PyObject *op*)
Return value: Borrowed reference. Return the annotations of the function object *op*. This can be a mutable dictionary or `NULL`.

int **PyFunction_SetAnnotations**(*PyObject *op*, *PyObject *annotations*)
Set the annotations for the function object *op*. *annotations* must be a dictionary or `Py_None`.

실패하면 `SystemError`를 발생시키고 `-1`을 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.

PyTypeObject **PyInstanceMethod_Type**
이 *PyTypeObject* 인스턴스는 파이썬 인스턴스 메서드 형을 나타냅니다. 파이썬 프로그램에는 노출되지 않습니다.

int **PyInstanceMethod_Check**(*PyObject *o*)
Return true if *o* is an instance method object (has type *PyInstanceMethod_Type*). The parameter must not be `NULL`.

*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.

PyTypeObject **PyMethod_Type**
이 *PyTypeObject* 인스턴스는 파이썬 메서드 형을 나타냅니다. 이것은 파이썬 프로그램에 `types.MethodType`로 노출됩니다.

int **PyMethod_Check**(*PyObject* *o)
Return true if *o* is a method object (has type *PyMethod_Type*). The parameter must not be `NULL`.

*PyObject** **PyMethod_New**(*PyObject* *func, *PyObject* *self)
Return value: New reference. Return a new method object, with *func* being any callable object and *self* the instance the method should be bound. *func* is the function that will be called when the method is called. *self* must not be `NULL`.

*PyObject** **PyMethod_Function**(*PyObject* *meth)
Return value: Borrowed reference. *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 역참조되지 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)
Return true if *ob* is a cell object; *ob* must not be `NULL`.

*PyObject** **PyCell_New**(*PyObject* *ob)
Return value: New reference. Create and return a new cell object containing the value *ob*. The parameter may be `NULL`.

*PyObject** **PyCell_Get**(*PyObject* *cell)
Return value: New reference. 셀 *cell*의 내용을 반환합니다.

*PyObject** **PyCell_GET**(*PyObject* *cell)
Return value: Borrowed reference. Return the contents of the cell *cell*, but without checking that *cell* is non-`NULL` and a cell object.

int **PyCell_Set**(*PyObject* *cell, *PyObject* *value)
Set the contents of the cell object *cell* to *value*. This releases the reference to any current content of the cell. *value* may be `NULL`. *cell* must be non-`NULL`; if it is not a cell object, `-1` will be returned. On success, `0` will be returned.

void **PyCell_SET**(*PyObject* *cell, *PyObject* *value)
Sets the value of the cell object *cell* to *value*. No reference counts are adjusted, and no checks are made for safety; *cell* must be non-NULL and must be a cell object.

8.5.5 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)
*co*가 code 객체면 참을 반환합니다.

int **PyCode_GetNumFree**(*PyCodeObject* *co)
*co*에 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* *lnotab)
Return value: New reference. 새 코드 객체를 반환합니다. 프레임을 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_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 에뮬레이션입니다. 파이썬 3에서, 파일과 스트림은 새로운 io 모듈을 사용합니다. 이 모듈은 운영 체제의 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.

PyFile_FromFd(int *fd*, const char *name, const char *mode, int *buffering*, const char *encoding, const char *errors, const char *newline, int *closefd*)
Return value: New reference. Create a Python file object from the file descriptor of an already opened file *fd*. The arguments *name*, *encoding*, *errors* and *newline* can be NULL to use the defaults; *buffering* can be *-1* to use the default. *name* is ignored and kept for backward compatibility. Return NULL on failure. For a more comprehensive description of the arguments, please refer to the io.open() function documentation.

> **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 객체

의 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([n])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 관계없이 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_WriteObject**(*PyObject *obj*, *PyObject *p*, int *flags*)
객체 *obj*를 파일 객체 *p*에 씁니다. *flags*에서 지원되는 유일한 플래그는 Py_PRINT_RAW입니다; 주어지면, 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 Module Objects

PyTypeObject **PyModule_Type**
This instance of *PyTypeObject* represents the Python module type. This is exposed to Python programs as types.ModuleType.

int **PyModule_Check**(*PyObject *p*)
Return true if *p* is a module object, or a subtype of a module object.

int **PyModule_CheckExact**(*PyObject *p*)
Return true if *p* is a module object, but not a subtype of *PyModule_Type*.

*PyObject** **PyModule_NewObject**(*PyObject *name*)
Return value: New reference. Return a new module object with the __name__ attribute set to *name*. The module's __name__, __doc__, __package__, and __loader__ attributes are filled in (all but __name__ are set to None); the caller is responsible for providing a __file__ attribute.

버전 3.3에 추가.

버전 3.4에서 변경: __package__ and __loader__ are set to None.

*PyObject** **PyModule_New**(const char *name)
Return value: New reference. Similar to *PyModule_NewObject()*, but the name is a UTF-8 encoded string instead of a Unicode object.

*PyObject** **PyModule_GetDict**(*PyObject *module*)
Return value: Borrowed reference. Return the dictionary object that implements *module*'s namespace; this object is the same as the __dict__ attribute of the module object. If *module* is not a module object (or a subtype of a module object), SystemError is raised and NULL is returned.

It is recommended extensions use other PyModule_*() and PyObject_*() functions rather than directly manipulate a module's __dict__.

*PyObject** **PyModule_GetNameObject**(*PyObject *module*)
Return value: New reference. Return *module*'s __name__ value. If the module does not provide one, or if it is not a string, SystemError is raised and NULL is returned.

버전 3.3에 추가.

const char* **PyModule_GetName**(*PyObject *module*)
Similar to *PyModule_GetNameObject()* but return the name encoded to 'utf-8'.

void* **PyModule_GetState**(*PyObject *module*)
Return the "state" of the module, that is, a pointer to the block of memory allocated at module creation time, or NULL. See *PyModuleDef.m_size*.

*PyModuleDef** **PyModule_GetDef**(*PyObject *module*)
Return a pointer to the *PyModuleDef* struct from which the module was created, or NULL if the module wasn't created from a definition.

*PyObject** **PyModule_GetFilenameObject**(*PyObject *module*)
Return value: New reference. Return the name of the file from which *module* was loaded using *module*'s __file__ attribute. If this is not defined, or if it is not a unicode string, raise SystemError and return NULL; otherwise return a reference to a Unicode object.

버전 3.2에 추가.

const char* **PyModule_GetFilename**(*PyObject *module*)
Similar to *PyModule_GetFilenameObject()* but return the filename encoded to 'utf-8'.

버전 3.2부터 폐지: *PyModule_GetFilename()* raises UnicodeEncodeError on unencodable filenames, use *PyModule_GetFilenameObject()* instead.

Initializing C modules

Modules objects are usually created from extension modules (shared libraries which export an initialization function), or compiled-in modules (where the initialization function is added using *PyImport_AppendInittab()*). See building or extending-with-embedding for details.

The initialization function can either pass a module definition instance to *PyModule_Create()*, and return the resulting module object, or request "multi-phase initialization" by returning the definition struct itself.

PyModuleDef
The module definition struct, which holds all information needed to create a module object. There is usually only one statically initialized variable of this type for each module.

PyModuleDef_Base **m_base**
Always initialize this member to PyModuleDef_HEAD_INIT.

const char ***m_name**
Name for the new module.

const char ***m_doc**
Docstring for the module; usually a docstring variable created with *PyDoc_STRVAR* is used.

Py_ssize_t **m_size**
Module state may be kept in a per-module memory area that can be retrieved with *PyModule_GetState()*, rather than in static globals. This makes modules safe for use in multiple sub-interpreters.

This memory area is allocated based on *m_size* on module creation, and freed when the module object is deallocated, after the m_free function has been called, if present.

Setting m_size to -1 means that the module does not support sub-interpreters, because it has global state.

Setting it to a non-negative value means that the module can be re-initialized and specifies the additional amount of memory it requires for its state. Non-negative m_size is required for multi-phase initialization.

See **PEP 3121** for more details.

*PyMethodDef** **m_methods**
A pointer to a table of module-level functions, described by *PyMethodDef* values. Can be NULL if no functions are present.

*PyModuleDef_Slot** **m_slots**
An array of slot definitions for multi-phase initialization, terminated by a {0, NULL} entry. When using single-phase initialization, *m_slots* must be NULL.

버전 3.5에서 변경: Prior to version 3.5, this member was always set to NULL, and was defined as:

inquiry **m_reload**

traverseproc **m_traverse**
A traversal function to call during GC traversal of the module object, or `NULL` if not needed. This function may be called before module state is allocated (*PyModule_GetState()* may return *NULL*), and before the *Py_mod_exec* function is executed.

inquiry **m_clear**
A clear function to call during GC clearing of the module object, or `NULL` if not needed. This function may be called before module state is allocated (*PyModule_GetState()* may return *NULL*), and before the *Py_mod_exec* function is executed.

freefunc **m_free**
A function to call during deallocation of the module object, or `NULL` if not needed. This function may be called before module state is allocated (*PyModule_GetState()* may return *NULL*), and before the *Py_mod_exec* function is executed.

Single-phase initialization

The module initialization function may create and return the module object directly. This is referred to as "single-phase initialization", and uses one of the following two module creation functions:

*PyObject** **PyModule_Create**(*PyModuleDef* *def*)
Return value: New reference. Create a new module object, given the definition in *def*. This behaves like *PyModule_Create2()* with *module_api_version* set to `PYTHON_API_VERSION`.

*PyObject** **PyModule_Create2**(*PyModuleDef* *def*, int *module_api_version*)
Return value: New reference. Create a new module object, given the definition in *def*, assuming the API version *module_api_version*. If that version does not match the version of the running interpreter, a `RuntimeWarning` is emitted.

참고: Most uses of this function should be using *PyModule_Create()* instead; only use this if you are sure you need it.

Before it is returned from in the initialization function, the resulting module object is typically populated using functions like *PyModule_AddObject()*.

Multi-phase initialization

An alternate way to specify extensions is to request "multi-phase initialization". Extension modules created this way behave more like Python modules: the initialization is split between the *creation phase*, when the module object is created, and the *execution phase*, when it is populated. The distinction is similar to the `__new__()` and `__init__()` methods of classes.

Unlike modules created using single-phase initialization, these modules are not singletons: if the *sys.modules* entry is removed and the module is re-imported, a new module object is created, and the old module is subject to normal garbage collection – as with Python modules. By default, multiple modules created from the same definition should be independent: changes to one should not affect the others. This means that all state should be specific to the module object (using e.g. using *PyModule_GetState()*), or its contents (such as the module's `__dict__` or individual classes created with *PyType_FromSpec()*).

All modules created using multi-phase initialization are expected to support *sub-interpreters*. Making sure multiple modules are independent is typically enough to achieve this.

To request multi-phase initialization, the initialization function (PyInit_modulename) returns a *PyModuleDef* instance with non-empty *m_slots*. Before it is returned, the `PyModuleDef` instance must be initialized with the following function:

*PyObject** **PyModuleDef_Init**(*PyModuleDef *def*)

Return value: Borrowed reference. Ensures a module definition is a properly initialized Python object that correctly reports its type and reference count.

Returns *def* cast to `PyObject*`, or `NULL` if an error occurred.

버전 3.5에 추가.

The *m_slots* member of the module definition must point to an array of `PyModuleDef_Slot` structures:

PyModuleDef_Slot

int **slot**
A slot ID, chosen from the available values explained below.

void* **value**
Value of the slot, whose meaning depends on the slot ID.

버전 3.5에 추가.

The *m_slots* array must be terminated by a slot with id 0.

The available slot types are:

Py_mod_create
Specifies a function that is called to create the module object itself. The *value* pointer of this slot must point to a function of the signature:

*PyObject** **create_module**(*PyObject *spec*, *PyModuleDef *def*)

The function receives a `ModuleSpec` instance, as defined in **PEP 451**, and the module definition. It should return a new module object, or set an error and return `NULL`.

This function should be kept minimal. In particular, it should not call arbitrary Python code, as trying to import the same module again may result in an infinite loop.

Multiple `Py_mod_create` slots may not be specified in one module definition.

If `Py_mod_create` is not specified, the import machinery will create a normal module object using *PyModule_New()*. The name is taken from *spec*, not the definition, to allow extension modules to dynamically adjust to their place in the module hierarchy and be imported under different names through symlinks, all while sharing a single module definition.

There is no requirement for the returned object to be an instance of *PyModule_Type*. Any type can be used, as long as it supports setting and getting import-related attributes. However, only `PyModule_Type` instances may be returned if the `PyModuleDef` has non-`NULL` `m_traverse`, `m_clear`, `m_free`; non-zero `m_size`; or slots other than `Py_mod_create`.

Py_mod_exec
Specifies a function that is called to *execute* the module. This is equivalent to executing the code of a Python module: typically, this function adds classes and constants to the module. The signature of the function is:

int **exec_module**(*PyObject* *module*)

If multiple `Py_mod_exec` slots are specified, they are processed in the order they appear in the *m_slots* array.

See **PEP 489** for more details on multi-phase initialization.

Low-level module creation functions

The following functions are called under the hood when using multi-phase initialization. They can be used directly, for example when creating module objects dynamically. Note that both `PyModule_FromDefAndSpec` and `PyModule_ExecDef` must be called to fully initialize a module.

PyObject * **PyModule_FromDefAndSpec**(*PyModuleDef* *def, *PyObject* *spec)
Return value: New reference. Create a new module object, given the definition in *module* and the ModuleSpec *spec*. This behaves like *PyModule_FromDefAndSpec2()* with *module_api_version* set to `PYTHON_API_VERSION`.

버전 3.5에 추가.

PyObject * **PyModule_FromDefAndSpec2**(*PyModuleDef* *def, *PyObject* *spec, int *module_api_version*)
Return value: New reference. Create a new module object, given the definition in *module* and the ModuleSpec *spec*, assuming the API version *module_api_version*. If that version does not match the version of the running interpreter, a `RuntimeWarning` is emitted.

참고: Most uses of this function should be using *PyModule_FromDefAndSpec()* instead; only use this if you are sure you need it.

버전 3.5에 추가.

int **PyModule_ExecDef**(*PyObject* *module, *PyModuleDef* *def)
Process any execution slots (*Py_mod_exec*) given in *def*.

버전 3.5에 추가.

int **PyModule_SetDocString**(*PyObject* *module, const char *docstring)
Set the docstring for *module* to *docstring*. This function is called automatically when creating a module from `PyModuleDef`, using either `PyModule_Create` or `PyModule_FromDefAndSpec`.

버전 3.5에 추가.

int **PyModule_AddFunctions**(*PyObject* *module, *PyMethodDef* *functions)
Add the functions from the `NULL` terminated *functions* array to *module*. Refer to the *PyMethodDef* documentation for details on individual entries (due to the lack of a shared module namespace, module level "functions" implemented in C typically receive the module as their first parameter, making them similar to instance methods on Python classes). This function is called automatically when creating a module from `PyModuleDef`, using either `PyModule_Create` or `PyModule_FromDefAndSpec`.

버전 3.5에 추가.

Support functions

The module initialization function (if using single phase initialization) or a function called from a module execution slot (if using multi-phase initialization), can use the following functions to help initialize the module state:

int **PyModule_AddObject**(*PyObject* *module, const char *name, *PyObject* *value)
Add an object to *module* as *name*. This is a convenience function which can be used from the module's initialization function. This steals a reference to *value* on success. Return `-1` on error, `0` on success.

참고: Unlike other functions that steal references, `PyModule_AddObject()` only decrements the reference count of *value* **on success**.

This means that its return value must be checked, and calling code must *Py_DECREF()* *value* manually on error. Example usage:

```
Py_INCREF(spam);
if (PyModule_AddObject(module, "spam", spam) < 0) {
    Py_DECREF(module);
    Py_DECREF(spam);
    return NULL;
}
```

int **PyModule_AddIntConstant**(*PyObject* *module, const char *name, long value)
Add an integer constant to *module* as *name*. This convenience function can be used from the module's initialization function. Return `-1` on error, `0` on success.

int **PyModule_AddStringConstant**(*PyObject* *module, const char *name, const char *value)
Add a string constant to *module* as *name*. This convenience function can be used from the module's initialization function. The string *value* must be `NULL`-terminated. Return `-1` on error, `0` on success.

int **PyModule_AddIntMacro**(*PyObject* *module, macro)
Add an int constant to *module*. The name and the value are taken from *macro*. For example `PyModule_AddIntMacro(module, AF_INET)` adds the int constant *AF_INET* with the value of *AF_INET* to *module*. Return `-1` on error, `0` on success.

int **PyModule_AddStringMacro**(*PyObject* *module, macro)
Add a string constant to *module*.

Module lookup

Single-phase initialization creates singleton modules that can be looked up in the context of the current interpreter. This allows the module object to be retrieved later with only a reference to the module definition.

These functions will not work on modules created using multi-phase initialization, since multiple such modules can be created from a single definition.

*PyObject** **PyState_FindModule**(*PyModuleDef* *def)
Return value: Borrowed reference. Returns the module object that was created from *def* for the current interpreter. This method requires that the module object has been attached to the interpreter state with *PyState_AddModule()* beforehand. In case the corresponding module object is not found or has not been attached to the interpreter state yet, it returns `NULL`.

int **PyState_AddModule**(*PyObject* *module, *PyModuleDef* *def)
Attaches the module object passed to the function to the interpreter state. This allows the module object to be accessible via *PyState_FindModule()*.

Only effective on modules created using single-phase initialization.

Python calls `PyState_AddModule` automatically after importing a module, so it is unnecessary (but harmless) to call it from module initialization code. An explicit call is needed only if the module's own init code subsequently calls `PyState_FindModule`. The function is mainly intended for implementing alternative import mechanisms (either by calling it directly, or by referring to its implementation for details of the required state updates).

Return 0 on success or -1 on failure.

버전 3.3에 추가.

int **PyState_RemoveModule**(*PyModuleDef* *def)
Removes the module object created from *def* from the interpreter state. Return 0 on success or -1 on failure.

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.

PyTypeObject **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)
*op*의 형이 *PySeqIter_Type*이면 참을 돌려줍니다.

*PyObject** **PySeqIter_New**(*PyObject *seq*)
Return value: New reference. 일반 시퀀스 객체 *seq*와 함께 작동하는 이터레이터를 반환합니다. 시퀀스가 서브스크립션 연산에서 `IndexError`를 일으키면 이터레이션이 끝납니다.

PyTypeObject **PyCallIter_Type**
*PyCallIter_New()*와 `iter()` 내장 함수의 두 인자 형식에 의해 반환된 이터레이터 객체에 대한 형 객체.

int **PyCallIter_Check**(op)
*op*의 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)
Return true if *ob* is a slice object; *ob* must not be `NULL`.

*PyObject** **PySlice_New**(*PyObject* *start, *PyObject* *stop, *PyObject* *step)
Return value: New reference. Return a new slice object with the given values. The *start*, *stop*, and *step* parameters are used as the values of the slice object attributes of the same names. Any of the values may be `NULL`, in which case the `None` will be used for the corresponding attribute. Return `NULL` if the new object could not be allocated.

int **PySlice_GetIndices**(*PyObject* *slice, Py_ssize_t *length*, Py_ssize_t *start*, Py_ssize_t *stop*, Py_ssize_t *step*)
길이가 *length*인 시퀀스를 가정하여, 슬라이스 객체 *slice*에서 start, stop 및 step 인덱스를 가져옵니다. *length*보다 큰 인덱스를 에러로 처리합니다.

성공하면 `0`을 반환하고, 에러면 예외 설정 없이 `-1`을 반환합니다(인덱스 중 하나가 `None`이 아니고 정수로 변환되지 않는 한, 이때는 예외를 설정하고 `-1`을 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_GetIndices()*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 `0`을 반환하고, 에러면 예외를 설정하고 `-1`을 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_MIN로 조용히 높이고, -PY_SSIZE_T_MAX보다 작은 step 값을 -PY_SSIZE_T_MAX로 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*과 마찬가지로 싱글 톤 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' ortran *order*로)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*)

Return either a pointer to the exporting object that the memoryview is based on or NULL if the memoryview has been created by one of the functions *PyMemoryView_FromMemory()* or *PyMemoryView_FromBuffer()*. *mview* **must** be a memoryview instance.

8.6.8 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. Return a weak reference object for the object *ob*. This will always return a new reference, but is not guaranteed to create a new object; an existing reference object may be returned. The second parameter, *callback*, can be a callable object that receives notification when *ob* is garbage collected; it should accept a single parameter, which will be the weak reference object itself. *callback* may also be `None` or `NULL`. If *ob* is not a weakly-referencable object, or if *callback* is not callable, `None`, or `NULL`, this will return `NULL` and raise `TypeError`.

*PyObject** **PyWeakref_NewProxy**(*PyObject *ob*, *PyObject *callback*)
Return value: New reference. Return a weak reference proxy object for the object *ob*. This will always return a new reference, but is not guaranteed to create a new object; an existing proxy object may be returned. The second parameter, *callback*, can be a callable object that receives notification when *ob* is garbage collected; it should accept a single parameter, which will be the weak reference object itself. *callback* may also be `None` or `NULL`. If *ob* is not a weakly-referencable object, or if *callback* is not callable, `None`, or `NULL`, this will return `NULL` and raise `TypeError`.

*PyObject** **PyWeakref_GetObject**(*PyObject *ref*)
Return value: Borrowed reference. 약한 참조(*ref*)로부터 참조된 객체를 반환합니다. 참조가 더는 살아 있지 않으면, `Py_None`을 반환합니다.

참고: 이 함수는 참조된 객체에 대한 **빌린 참조**를 반환합니다. 이는 객체를 계속 사용하는 동안 객체가 파괴될 수 없음을 알고 있을 때를 제외하고, 객체에 대해 항상 *Py_INCREF()*를 호출해야 함을 뜻합니다.

---

*PyObject** **PyWeakref_GET_OBJECT**(*PyObject *ref*)
Return value: Borrowed reference. *PyWeakref_GetObject()*와 유사하지만, 에러 검사를 수행하지 않는 매크로로 구현됩니다.

8.6.9 캡슐

이 객체 사용에 대한 자세한 정보는 using-capsules를 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 PyCapsule이면 참을 돌려줍니다.

*PyObject** **PyCapsule_New**(void *pointer, const char *name, *PyCapsule_Destructor* destructor)
Return value: New reference. Create a PyCapsule encapsulating the *pointer*. The *pointer* argument may not be NULL.

On failure, set an exception and return NULL.

The *name* string may either be NULL or a pointer to a valid C string. If non-NULL, this string must outlive the capsule. (Though it is permitted to free it inside the *destructor*.)

If the *destructor* argument is not NULL, it will be called with the capsule as its argument when it is destroyed.

이 캡슐을 모듈의 어트리뷰트로 저장하려면, *name*을 modulename.attributename로 지정해야 합니다. 이렇게 하면 다른 모듈이 PyCapsule_Import()를 사용하여 캡슐을 임포트 할 수 있습니다.

void* **PyCapsule_GetPointer**(*PyObject* *capsule, const char *name)
Retrieve the *pointer* stored in the capsule. On failure, set an exception and return NULL.

The *name* parameter must compare exactly to the name stored in the capsule. If the name stored in the capsule is NULL, the *name* passed in must also be NULL. Python uses the C function strcmp() to compare capsule names.

PyCapsule_Destructor **PyCapsule_GetDestructor**(*PyObject* *capsule)
Return the current destructor stored in the capsule. On failure, set an exception and return NULL.

It is legal for a capsule to have a NULL destructor. This makes a NULL return code somewhat ambiguous; use PyCapsule_IsValid() or PyErr_Occurred() to disambiguate.

void* **PyCapsule_GetContext**(*PyObject* *capsule)
Return the current context stored in the capsule. On failure, set an exception and return NULL.

It is legal for a capsule to have a NULL context. This makes a NULL return code somewhat ambiguous; use PyCapsule_IsValid() or PyErr_Occurred() to disambiguate.

const char* **PyCapsule_GetName**(*PyObject* *capsule)
Return the current name stored in the capsule. On failure, set an exception and return NULL.

It is legal for a capsule to have a NULL name. This makes a NULL return code somewhat ambiguous; use PyCapsule_IsValid() or PyErr_Occurred() to disambiguate.

void* **PyCapsule_Import**(const char *name, int *no_block*)
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).

Return the capsule's internal *pointer* on success. On failure, set an exception and return NULL.

int **PyCapsule_IsValid**(*PyObject* *capsule, const char *name)
Determines whether or not *capsule* is a valid capsule. A valid capsule is non-NULL, passes PyCapsule_CheckExact(), has a non-NULL pointer stored in it, and its internal name matches the *name* parameter. (See PyCapsule_GetPointer() for information on how capsule names are compared.)

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*)
Set the name inside *capsule* to *name*. If non-NULL, the name must outlive the capsule. If the previous *name* stored in the capsule was not NULL, no attempt is made to free it.

성공하면 0을 반환합니다. 실패하면 0이 아닌 값을 반환하고 예외를 설정합니다.

int **PyCapsule_SetPointer**(*PyObject* *capsule*, void *pointer*)
Set the void pointer inside *capsule* to *pointer*. The pointer may not be NULL.

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*)
Return true if *ob* is a generator object; *ob* must not be NULL.

int **PyGen_CheckExact**(*PyObject* *ob*)
Return true if *ob*'s type is *PyGen_Type*; *ob* must not be NULL.

*PyObject** **PyGen_New**(*PyFrameObject* *frame*)
Return value: New reference. Create and return a new generator object based on the *frame* object. A reference to *frame* is stolen by this function. The argument must not be NULL.

*PyObject** **PyGen_NewWithQualName**(*PyFrameObject* *frame*, *PyObject* *name*, *PyObject* *qualname*)
Return value: New reference. Create and return a new generator object based on the *frame* object, with __name__ and __qualname__ set to *name* and *qualname*. A reference to *frame* is stolen by this function. The *frame* argument must not be NULL.

8.6.11 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*)
Return true if *ob*'s type is *PyCoro_Type*; *ob* must not be NULL.

*PyObject** **PyCoro_New**(*PyFrameObject* *frame*, *PyObject* *name*, *PyObject* *qualname*)
Return value: New reference. Create and return a new coroutine object based on the *frame* object, with __name__ and __qualname__ set to *name* and *qualname*. A reference to *frame* is stolen by this function. The *frame* argument must not be NULL.

8.6.12 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-34762를 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)
Return true if *o* is of type *PyContext_Type*. *o* must not be `NULL`. This function always succeeds.

int **PyContextVar_CheckExact**(*PyObject* *o)
Return true if *o* is of type *PyContextVar_Type*. *o* must not be `NULL`. This function always succeeds.

int **PyContextToken_CheckExact**(*PyObject* *o)
Return true if *o* is of type *PyContextToken_Type*. *o* must not be `NULL`. This function always succeeds.

컨텍스트 객체 관리 함수:

PyObject ***PyContext_New**(void)
Return value: New reference. 새로운 빈 컨텍스트 객체를 만듭니다. 에러가 발생하면 `NULL`를 반환합니다.

PyObject ***PyContext_Copy**(*PyObject* *ctx)
Return value: New reference. 전달된 *ctx* 컨텍스트 객체의 얕은 복사본을 만듭니다. 에러가 발생하면 `NULL`을 반환합니다.

PyObject ***PyContext_CopyCurrent**(void)
Return value: New reference. 현재 스레드 컨텍스트의 얕은 복사본을 만듭니다. 에러가 발생하면 `NULL`을 반환합니다.

int **PyContext_Enter**(*PyObject* *ctx)
현재 스레드의 현재 컨텍스트로 *ctx*를 설정합니다. 성공 시 `0`을 반환하고, 에러 시 `-1`을 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. 새 ContextVar 객체를 만듭니다. *name* 매개 변수는 인트로스펙션과 디버그 목적으로 사용됩니다. *def* 매개 변수는 선택적으로 컨텍스트 변수의 기본값을 지정할 수 있습니다. 에러가 발생하면, 이 함수는 NULL을 반환합니다.

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, *value*는 그것을 가리키는 포인터가 됩니다. 컨텍스트 변수가 발견되지 않으면, *value*는 다음을 가리 킵니다:

- *default_value*, NULL이 아니면;
- *var*의 기본값, NULL이 아니면;
- NULL

값이 발견되면, 이 함수는 그것에 대한 새 참조를 만듭니다.

PyObject ***PyContextVar_Set**(*PyObject* *var, *PyObject* *value)
Return value: New reference. 현재 컨텍스트에서 *var*의 값을 *value*로 설정합니다. *PyObject* 객체에 대한 포인터를 반환하거나, 에러가 발생하면 NULL을 반환합니다.

int **PyContextVar_Reset**(*PyObject* *var, *PyObject* *token)
var 컨텍스트 변수의 상태를 *token*을 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)
Return true if *ob* is of type PyDateTime_DateType or a subtype of PyDateTime_DateType. *ob* must not be NULL.

int **PyDate_CheckExact**(*PyObject* *ob)
Return true if *ob* is of type PyDateTime_DateType. *ob* must not be NULL.

int **PyDateTime_Check**(*PyObject* *ob)
Return true if *ob* is of type PyDateTime_DateTimeType or a subtype of PyDateTime_DateTimeType. *ob* must not be NULL.

int **PyDateTime_CheckExact**(*PyObject* *ob)
Return true if *ob* is of type PyDateTime_DateTimeType. *ob* must not be NULL.

int **PyTime_Check**(*PyObject* *ob)
Return true if *ob* is of type `PyDateTime_TimeType` or a subtype of `PyDateTime_TimeType`. *ob* must not be `NULL`.

int **PyTime_CheckExact**(*PyObject* *ob)
Return true if *ob* is of type `PyDateTime_TimeType`. *ob* must not be `NULL`.

int **PyDelta_Check**(*PyObject* *ob)
Return true if *ob* is of type `PyDateTime_DeltaType` or a subtype of `PyDateTime_DeltaType`. *ob* must not be `NULL`.

int **PyDelta_CheckExact**(*PyObject* *ob)
Return true if *ob* is of type `PyDateTime_DeltaType`. *ob* must not be `NULL`.

int **PyTZInfo_Check**(*PyObject* *ob)
Return true if *ob* is of type `PyDateTime_TZInfoType` or a subtype of `PyDateTime_TZInfoType`. *ob* must not be `NULL`.

int **PyTZInfo_CheckExact**(*PyObject* *ob)
Return true if *ob* is of type `PyDateTime_TZInfoType`. *ob* must not be `NULL`.

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 관해 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.

Macros to extract fields from date objects. The argument must be an instance of `PyDateTime_Date`, including subclasses (such as `PyDateTime_DateTime`). The argument must not be `NULL`, and the type is not checked:

int **PyDateTime_GET_YEAR**(PyDateTime_Date *o)
양의 int로, 년을 반환합니다.

int **PyDateTime_GET_MONTH**(PyDateTime_Date *o)
1에서 12까지의 int로, 월을 반환합니다.

int **PyDateTime_GET_DAY**(PyDateTime_Date *o)
1에서 31까지의 int로, 일을 반환합니다.

Macros to extract fields from datetime objects. The argument must be an instance of `PyDateTime_DateTime`, including subclasses. The argument must not be `NULL`, and the type is not checked:

int **PyDateTime_DATE_GET_HOUR**(PyDateTime_DateTime *o)
0부터 23까지의 int로, 시를 반환합니다.

int **PyDateTime_DATE_GET_MINUTE**(PyDateTime_DateTime *o)
0부터 59까지의 int로, 분을 반환합니다.

int **PyDateTime_DATE_GET_SECOND**(PyDateTime_DateTime *o)
0부터 59까지의 int로, 초를 반환합니다.

int **PyDateTime_DATE_GET_MICROSECOND**(PyDateTime_DateTime *o)
0부터 999999까지의 int로, 마이크로초를 반환합니다.

int **PyDateTime_DATE_GET_FOLD**(PyDateTime_DateTime *o)
Return the fold, as an int from 0 through 1.

버전 3.6에 추가.

Macros to extract fields from time objects. The argument must be an instance of `PyDateTime_Time`, including subclasses. The argument must not be `NULL`, and the type is not checked:

int **PyDateTime_TIME_GET_HOUR**(PyDateTime_Time *o)
0부터 23까지의 int로, 시를 반환합니다.

int **PyDateTime_TIME_GET_MINUTE**(PyDateTime_Time *o)
0부터 59까지의 int로, 분을 반환합니다.

int **PyDateTime_TIME_GET_SECOND**(PyDateTime_Time *o)
0부터 59까지의 int로, 초를 반환합니다.

int **PyDateTime_TIME_GET_MICROSECOND**(PyDateTime_Time *o)
0부터 999999까지의 int로, 마이크로초를 반환합니다.

int **PyDateTime_TIME_GET_FOLD**(PyDateTime_Time *o)
Return the fold, as an int from 0 through 1.

버전 3.6에 추가.

Macros to extract fields from time delta objects. The argument must be an instance of `PyDateTime_Delta`, including subclasses. The argument must not be `NULL`, and the type is not checked:

int **PyDateTime_DELTA_GET_DAYS**(PyDateTime_Delta *o)
-999999999에서 999999999까지의 int로, 일 수를 반환합니다.

버전 3.3에 추가.

int **PyDateTime_DELTA_GET_SECONDS**(PyDateTime_Delta *o)
0부터 86399까지의 int로, 초 수를 반환합니다.

버전 3.3에 추가.

int **PyDateTime_DELTA_GET_MICROSECONDS**(PyDateTime_Delta *o)
0에서 999999까지의 int로, 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.

CHAPTER 9

Initialization, Finalization, and Threads

9.1 Before Python Initialization

In an application embedding Python, the *Py_Initialize()* function must be called before using any other Python/C API functions; with the exception of a few functions and the *global configuration variables*.

The following functions can be safely called before Python is initialized:

- Configuration functions:
  - *PyImport_AppendInittab()*
  - *PyImport_ExtendInittab()*
  - PyInitFrozenExtensions()
  - *PyMem_SetAllocator()*
  - *PyMem_SetupDebugHooks()*
  - *PyObject_SetArenaAllocator()*
  - *Py_SetPath()*
  - *Py_SetProgramName()*
  - *Py_SetPythonHome()*
  - *Py_SetStandardStreamEncoding()*
  - *PySys_AddWarnOption()*
  - *PySys_AddXOption()*
  - *PySys_ResetWarnOptions()*
- Informative functions:
  - *Py_IsInitialized()*
  - *PyMem_GetAllocator()*
  - *PyObject_GetArenaAllocator()*
  - *Py_GetBuildInfo()*
  - *Py_GetCompiler()*

  - *Py_GetCopyright()*
  - *Py_GetPlatform()*
  - *Py_GetVersion()*
- Utilities:
  - *Py_DecodeLocale()*
- Memory allocators:
  - *PyMem_RawMalloc()*
  - *PyMem_RawRealloc()*
  - *PyMem_RawCalloc()*
  - *PyMem_RawFree()*

참고: The following functions **should not be called** before *Py_Initialize()*: *Py_EncodeLocale()*, *Py_GetPath()*, *Py_GetPrefix()*, *Py_GetExecPrefix()*, *Py_GetProgramFullPath()*, *Py_GetPythonHome()*, *Py_GetProgramName()* and *PyEval_InitThreads()*.

9.2 Global configuration variables

Python has variables for the global configuration to control different features and options. By default, these flags are controlled by command line options.

When a flag is set by an option, the value of the flag is the number of times that the option was set. For example, `-b` sets *Py_BytesWarningFlag* to 1 and `-bb` sets *Py_BytesWarningFlag* to 2.

Py_BytesWarningFlag
Issue a warning when comparing `bytes` or `bytearray` with `str` or `bytes` with `int`. Issue an error if greater or equal to `2`.

Set by the `-b` option.

Py_DebugFlag
Turn on parser debugging output (for expert only, depending on compilation options).

Set by the `-d` option and the `PYTHONDEBUG` environment variable.

Py_DontWriteBytecodeFlag
If set to non-zero, Python won't try to write `.pyc` files on the import of source modules.

Set by the `-B` option and the `PYTHONDONTWRITEBYTECODE` environment variable.

Py_FrozenFlag
Suppress error messages when calculating the module search path in *Py_GetPath()*.

Private flag used by `_freeze_importlib` and `frozenmain` programs.

Py_HashRandomizationFlag
Set to `1` if the `PYTHONHASHSEED` environment variable is set to a non-empty string.

If the flag is non-zero, read the `PYTHONHASHSEED` environment variable to initialize the secret hash seed.

Py_IgnoreEnvironmentFlag
Ignore all `PYTHON*` environment variables, e.g. `PYTHONPATH` and `PYTHONHOME`, that might be set.

Set by the `-E` and `-I` options.

Py_InspectFlag
When a script is passed as first argument or the `-c` option is used, enter interactive mode after executing the script or the command, even when `sys.stdin` does not appear to be a terminal.

Set by the `-i` option and the `PYTHONINSPECT` environment variable.

Py_InteractiveFlag
Set by the `-i` option.

Py_IsolatedFlag
Run Python in isolated mode. In isolated mode `sys.path` contains neither the script's directory nor the user's site-packages directory.

Set by the `-I` option.

버전 3.4에 추가.

Py_LegacyWindowsFSEncodingFlag
If the flag is non-zero, use the `mbcs` encoding instead of the UTF-8 encoding for the filesystem encoding.

Set to `1` if the `PYTHONLEGACYWINDOWSFSENCODING` environment variable is set to a non-empty string.

See **PEP 529** for more details.

Availability: Windows.

Py_LegacyWindowsStdioFlag
If the flag is non-zero, use `io.FileIO` instead of `WindowsConsoleIO` for `sys` standard streams.

Set to `1` if the `PYTHONLEGACYWINDOWSSTDIO` environment variable is set to a non-empty string.

See **PEP 528** for more details.

Availability: Windows.

Py_NoSiteFlag
Disable the import of the module `site` and the site-dependent manipulations of `sys.path` that it entails. Also disable these manipulations if `site` is explicitly imported later (call `site.main()` if you want them to be triggered).

Set by the `-S` option.

Py_NoUserSiteDirectory
Don't add the `user site-packages directory` to `sys.path`.

Set by the `-s` and `-I` options, and the `PYTHONNOUSERSITE` environment variable.

Py_OptimizeFlag
Set by the `-O` option and the `PYTHONOPTIMIZE` environment variable.

Py_QuietFlag
Don't display the copyright and version messages even in interactive mode.

Set by the `-q` option.

버전 3.2에 추가.

Py_UnbufferedStdioFlag
Force the stdout and stderr streams to be unbuffered.

Set by the `-u` option and the `PYTHONUNBUFFERED` environment variable.

Py_VerboseFlag
Print a message each time a module is initialized, showing the place (filename or built-in module) from which it is loaded. If greater or equal to `2`, print a message for each file that is checked for when searching for a module. Also provides information on module cleanup at exit.

Set by the `-v` option and the `PYTHONVERBOSE` environment variable.

9.3 Initializing and finalizing the interpreter

void **Py_Initialize**()
Initialize the Python interpreter. In an application embedding Python, this should be called before using any other Python/C API functions; see *Before Python Initialization* for the few exceptions.

This initializes the table of loaded modules (`sys.modules`), and creates the fundamental modules `builtins`, `__main__` and `sys`. It also initializes the module search path (`sys.path`). It does not set `sys.argv`; use *PySys_SetArgvEx()* for that. This is a no-op when called for a second time (without calling *Py_FinalizeEx()* first). There is no return value; it is a fatal error if the initialization fails.

참고: On Windows, changes the console mode from `O_TEXT` to `O_BINARY`, which will also affect non-Python uses of the console using the C Runtime.

---

void **Py_InitializeEx**(int *initsigs*)
This function works like *Py_Initialize()* if *initsigs* is `1`. If *initsigs* is `0`, it skips initialization registration of signal handlers, which might be useful when Python is embedded.

int **Py_IsInitialized**()
Return true (nonzero) when the Python interpreter has been initialized, false (zero) if not. After *Py_FinalizeEx()* is called, this returns false until *Py_Initialize()* is called again.

int **Py_FinalizeEx**()
Undo all initializations made by *Py_Initialize()* and subsequent use of Python/C API functions, and destroy all sub-interpreters (see *Py_NewInterpreter()* below) that were created and not yet destroyed since the last call to *Py_Initialize()*. Ideally, this frees all memory allocated by the Python interpreter. This is a no-op when called for a second time (without calling *Py_Initialize()* again first). Normally the return value is `0`. If there were errors during finalization (flushing buffered data), `-1` is returned.

This function is provided for a number of reasons. An embedding application might want to restart Python without having to restart the application itself. An application that has loaded the Python interpreter from a dynamically loadable library (or DLL) might want to free all memory allocated by Python before unloading the DLL. During a hunt for memory leaks in an application a developer might want to free all memory allocated by Python before exiting from the application.

Bugs and caveats: The destruction of modules and objects in modules is done in random order; this may cause destructors (`__del__()` methods) to fail when they depend on other objects (even functions) or modules. Dynamically loaded extension modules loaded by Python are not unloaded. Small amounts of memory allocated by the Python interpreter may not be freed (if you find a leak, please report it). Memory tied up in circular references between objects is not freed. Some memory allocated by extension modules may not be freed. Some extensions may not work properly if their initialization routine is called more than once; this can happen if an application calls *Py_Initialize()* and *Py_FinalizeEx()* more than once.

버전 3.6에 추가.

void **Py_Finalize**()
This is a backwards-compatible version of *Py_FinalizeEx()* that disregards the return value.

9.4 Process-wide parameters

int **Py_SetStandardStreamEncoding**(const char **encoding*, const char **errors*)
This function should be called before *Py_Initialize()*, if it is called at all. It specifies which encoding and error handling to use with standard IO, with the same meanings as in `str.encode()`.

It overrides `PYTHONIOENCODING` values, and allows embedding code to control IO encoding when the environment variable does not work.

encoding and/or *errors* may be `NULL` to use `PYTHONIOENCODING` and/or default values (depending on other settings).

Note that `sys.stderr` always uses the "backslashreplace" error handler, regardless of this (or any other) setting.

If *Py_FinalizeEx()* is called, this function will need to be called again in order to affect subsequent calls to *Py_Initialize()*.

Returns `0` if successful, a nonzero value on error (e.g. calling after the interpreter has already been initialized).

버전 3.4에 추가.

void **Py_SetProgramName**(const wchar_t **name*)
This function should be called before *Py_Initialize()* is called for the first time, if it is called at all. It tells the interpreter the value of the `argv[0]` argument to the `main()` function of the program (converted to wide characters). This is used by *Py_GetPath()* and some other functions below to find the Python run-time libraries relative to the interpreter executable. The default value is `'python'`. The argument should point to a zero-terminated wide character string in static storage whose contents will not change for the duration of the program's execution. No code in the Python interpreter will change the contents of this storage.

Use *Py_DecodeLocale()* to decode a bytes string to get a `wchar_*` string.

wchar* **Py_GetProgramName**()
Return the program name set with *Py_SetProgramName()*, or the default. The returned string points into static storage; the caller should not modify its value.

wchar_t* **Py_GetPrefix**()
Return the *prefix* for installed platform-independent files. This is derived through a number of complicated rules from the program name set with *Py_SetProgramName()* and some environment variables; for example, if the program name is `'/usr/local/bin/python'`, the prefix is `'/usr/local'`. The returned string points into static storage; the caller should not modify its value. This corresponds to the **prefix** variable in the top-level `Makefile` and the `--prefix` argument to the **configure** script at build time. The value is available to Python code as `sys.prefix`. It is only useful on Unix. See also the next function.

wchar_t* **Py_GetExecPrefix**()
Return the *exec-prefix* for installed platform-*dependent* files. This is derived through a number of complicated rules from the program name set with *Py_SetProgramName()* and some environment variables; for example, if the program name is `'/usr/local/bin/python'`, the exec-prefix is `'/usr/local'`. The returned string points into static storage; the caller should not modify its value. This corresponds to the **exec_prefix** variable in the top-level `Makefile` and the `--exec-prefix` argument to the **configure** script at build time. The value is available to Python code as `sys.exec_prefix`. It is only useful on Unix.

Background: The exec-prefix differs from the prefix when platform dependent files (such as executables and shared libraries) are installed in a different directory tree. In a typical installation, platform dependent files may be installed in the `/usr/local/plat` subtree while platform independent may be installed in `/usr/local`.

Generally speaking, a platform is a combination of hardware and software families, e.g. Sparc machines running the Solaris 2.x operating system are considered the same platform, but Intel machines running Solaris 2.x are another platform, and Intel machines running Linux are yet another platform. Different major revisions of the same operating system generally also form different platforms. Non-Unix operating systems are a different story; the installation strategies on those systems are so different that the prefix and exec-prefix are meaningless, and set to the empty string. Note that compiled Python bytecode files are platform independent (but not independent from the Python version by which they were compiled!).

System administrators will know how to configure the **mount** or **automount** programs to share `/usr/local` between platforms while having `/usr/local/plat` be a different filesystem for each platform.

wchar_t* **Py_GetProgramFullPath**()
Return the full program name of the Python executable; this is computed as a side-effect of deriving the default module search path from the program name (set by *Py_SetProgramName()* above). The returned string points into static storage; the caller should not modify its value. The value is available to Python code as `sys.executable`.

wchar_t* **Py_GetPath**()
Return the default module search path; this is computed from the program name (set by

Py_SetProgramName() above) and some environment variables. The returned string consists of a series of directory names separated by a platform dependent delimiter character. The delimiter character is `':'` on Unix and Mac OS X, `';'` on Windows. The returned string points into static storage; the caller should not modify its value. The list `sys.path` is initialized with this value on interpreter startup; it can be (and usually is) modified later to change the search path for loading modules.

void **Py_SetPath**(const wchar_t *)

Set the default module search path. If this function is called before *Py_Initialize()*, then *Py_GetPath()* won't attempt to compute a default search path but uses the one provided instead. This is useful if Python is embedded by an application that has full knowledge of the location of all modules. The path components should be separated by the platform dependent delimiter character, which is `':'` on Unix and Mac OS X, `';'` on Windows.

This also causes `sys.executable` to be set only to the raw program name (see *Py_SetProgramName()*) and for `sys.prefix` and `sys.exec_prefix` to be empty. It is up to the caller to modify these if required after calling *Py_Initialize()*.

Use *Py_DecodeLocale()* to decode a bytes string to get a `wchar_*` string.

The path argument is copied internally, so the caller may free it after the call completes.

const char* **Py_GetVersion**()

Return the version of this Python interpreter. This is a string that looks something like

```
"3.0a5+ (py3k:63103M, May 12 2008, 00:53:55) \n[GCC 4.2.3]"
```

The first word (up to the first space character) is the current Python version; the first three characters are the major and minor version separated by a period. The returned string points into static storage; the caller should not modify its value. The value is available to Python code as `sys.version`.

const char* **Py_GetPlatform**()

Return the platform identifier for the current platform. On Unix, this is formed from the "official" name of the operating system, converted to lower case, followed by the major revision number; e.g., for Solaris 2.x, which is also known as SunOS 5.x, the value is `'sunos5'`. On Mac OS X, it is `'darwin'`. On Windows, it is `'win'`. The returned string points into static storage; the caller should not modify its value. The value is available to Python code as `sys.platform`.

const char* **Py_GetCopyright**()

Return the official copyright string for the current Python version, for example

`'Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam'`

The returned string points into static storage; the caller should not modify its value. The value is available to Python code as `sys.copyright`.

const char* **Py_GetCompiler**()

Return an indication of the compiler used to build the current Python version, in square brackets, for example:

```
"[GCC 2.7.2.2]"
```

The returned string points into static storage; the caller should not modify its value. The value is available to Python code as part of the variable `sys.version`.

const char* **Py_GetBuildInfo**()

Return information about the sequence number and build date and time of the current Python interpreter instance, for example

```
"#67, Aug  1 1997, 22:34:28"
```

The returned string points into static storage; the caller should not modify its value. The value is available to Python code as part of the variable `sys.version`.

void **PySys_SetArgvEx**(int *argc*, wchar_t **argv*, int *updatepath*)

Set `sys.argv` based on *argc* and *argv*. These parameters are similar to those passed to the program's `main()` function with the difference that the first entry should refer to the script file to be executed rather

than the executable hosting the Python interpreter. If there isn't a script that will be run, the first entry in *argv* can be an empty string. If this function fails to initialize `sys.argv`, a fatal condition is signalled using *Py_FatalError()*.

If *updatepath* is zero, this is all the function does. If *updatepath* is non-zero, the function also modifies `sys.path` according to the following algorithm:

- If the name of an existing script is passed in `argv[0]`, the absolute path of the directory where the script is located is prepended to `sys.path`.
- Otherwise (that is, if *argc* is `0` or `argv[0]` doesn't point to an existing file name), an empty string is prepended to `sys.path`, which is the same as prepending the current working directory (`"."`).

Use *Py_DecodeLocale()* to decode a bytes string to get a `wchar_*` string.

참고: It is recommended that applications embedding the Python interpreter for purposes other than executing a single script pass `0` as *updatepath*, and update `sys.path` themselves if desired. See CVE-2008-5983.

On versions before 3.1.3, you can achieve the same effect by manually popping the first `sys.path` element after having called *PySys_SetArgv()*, for example using:

```
PyRun_SimpleString("import sys; sys.path.pop(0)\n");
```

버전 3.1.3에 추가.

void **PySys_SetArgv**(int *argc*, wchar_t **argv*)

This function works like *PySys_SetArgvEx()* with *updatepath* set to `1` unless the **python** interpreter was started with the `-I`.

Use *Py_DecodeLocale()* to decode a bytes string to get a `wchar_*` string.

버전 3.4에서 변경: The *updatepath* value depends on `-I`.

void **Py_SetPythonHome**(const wchar_t **home*)

Set the default "home" directory, that is, the location of the standard Python libraries. See `PYTHONHOME` for the meaning of the argument string.

The argument should point to a zero-terminated character string in static storage whose contents will not change for the duration of the program's execution. No code in the Python interpreter will change the contents of this storage.

Use *Py_DecodeLocale()* to decode a bytes string to get a `wchar_*` string.

w_char* **Py_GetPythonHome**()

Return the default "home", that is, the value set by a previous call to *Py_SetPythonHome()*, or the value of the `PYTHONHOME` environment variable if it is set.

9.5 Thread State and the Global Interpreter Lock

The Python interpreter is not fully thread-safe. In order to support multi-threaded Python programs, there's a global lock, called the *global interpreter lock* or *GIL*, that must be held by the current thread before it can safely access Python objects. Without the lock, even the simplest operations could cause problems in a multi-threaded program: for example, when two threads simultaneously increment the reference count of the same object, the reference count could end up being incremented only once instead of twice.

Therefore, the rule exists that only the thread that has acquired the *GIL* may operate on Python objects or call Python/C API functions. In order to emulate concurrency of execution, the interpreter regularly tries to switch threads (see `sys.setswitchinterval()`). The lock is also released around potentially blocking I/O operations like reading or writing a file, so that other Python threads can run in the meantime.

The Python interpreter keeps some thread-specific bookkeeping information inside a data structure called *PyThreadState*. There's also one global variable pointing to the current *PyThreadState*: it can be retrieved using *PyThreadState_Get()*.

9.5.1 Releasing the GIL from extension code

Most extension code manipulating the *GIL* has the following simple structure:

```
Save the thread state in a local variable.
Release the global interpreter lock.
... Do some blocking I/O operation ...
Reacquire the global interpreter lock.
Restore the thread state from the local variable.
```

This is so common that a pair of macros exists to simplify it:

```
Py_BEGIN_ALLOW_THREADS
... Do some blocking I/O operation ...
Py_END_ALLOW_THREADS
```

The *Py_BEGIN_ALLOW_THREADS* macro opens a new block and declares a hidden local variable; the *Py_END_ALLOW_THREADS* macro closes the block.

The block above expands to the following code:

```
PyThreadState *_save;

_save = PyEval_SaveThread();
... Do some blocking I/O operation ...
PyEval_RestoreThread(_save);
```

Here is how these functions work: the global interpreter lock is used to protect the pointer to the current thread state. When releasing the lock and saving the thread state, the current thread state pointer must be retrieved before the lock is released (since another thread could immediately acquire the lock and store its own thread state in the global variable). Conversely, when acquiring the lock and restoring the thread state, the lock must be acquired before storing the thread state pointer.

참고: Calling system I/O functions is the most common use case for releasing the GIL, but it can also be useful before calling long-running computations which don't need access to Python objects, such as compression or cryptographic functions operating over memory buffers. For example, the standard `zlib` and `hashlib` modules release the GIL when compressing or hashing data.

9.5.2 Non-Python created threads

When threads are created using the dedicated Python APIs (such as the `threading` module), a thread state is automatically associated to them and the code showed above is therefore correct. However, when threads are created from C (for example by a third-party library with its own thread management), they don't hold the GIL, nor is there a thread state structure for them.

If you need to call Python code from these threads (often this will be part of a callback API provided by the aforementioned third-party library), you must first register these threads with the interpreter by creating a thread state data structure, then acquiring the GIL, and finally storing their thread state pointer, before you can start using the Python/C API. When you are done, you should reset the thread state pointer, release the GIL, and finally free the thread state data structure.

The *PyGILState_Ensure()* and *PyGILState_Release()* functions do all of the above automatically. The typical idiom for calling into Python from a C thread is:

```
PyGILState_STATE gstate;
gstate = PyGILState_Ensure();

/* Perform Python actions here. */
result = CallSomeFunction();
/* evaluate result or handle exception */

/* Release the thread. No Python API allowed beyond this point. */
PyGILState_Release(gstate);
```

Note that the `PyGILState_*()` functions assume there is only one global interpreter (created automatically by *Py_Initialize()*). Python supports the creation of additional interpreters (using *Py_NewInterpreter()*), but mixing multiple interpreters and the `PyGILState_*()` API is unsupported.

Another important thing to note about threads is their behaviour in the face of the C `fork()` call. On most systems with `fork()`, after a process forks only the thread that issued the fork will exist. That also means any locks held by other threads will never be released. Python solves this for `os.fork()` by acquiring the locks it uses internally before the fork, and releasing them afterwards. In addition, it resets any lock-objects in the child. When extending or embedding Python, there is no way to inform Python of additional (non-Python) locks that need to be acquired before or reset after a fork. OS facilities such as `pthread_atfork()` would need to be used to accomplish the same thing. Additionally, when extending or embedding Python, calling `fork()` directly rather than through `os.fork()` (and returning to or calling into Python) may result in a deadlock by one of Python's internal locks being held by a thread that is defunct after the fork. *PyOS_AfterFork_Child()* tries to reset the necessary locks, but is not always able to.

9.5.3 High-level API

These are the most commonly used types and functions when writing C extension code, or when embedding the Python interpreter:

PyInterpreterState
: This data structure represents the state shared by a number of cooperating threads. Threads belonging to the same interpreter share their module administration and a few other internal items. There are no public members in this structure.

  Threads belonging to different interpreters initially share nothing, except process state like available memory, open file descriptors and such. The global interpreter lock is also shared by all threads, regardless of to which interpreter they belong.

PyThreadState
: This data structure represents the state of a single thread. The only public data member is *PyInterpreterState* `*interp`, which points to this thread's interpreter state.

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: This function is now called by *Py_Initialize()*, so you don't have to call it yourself anymore.

  버전 3.2에서 변경: This function cannot be called before *Py_Initialize()* anymore.

int **PyEval_ThreadsInitialized**()
: Returns a non-zero value if *PyEval_InitThreads()* has been called. This function can be called without holding the GIL, and therefore can be used to avoid calls to the locking API when running single-threaded.

  버전 3.7에서 변경: The *GIL* is now initialized by *Py_Initialize()*.

*PyThreadState** **PyEval_SaveThread**()
: Release the global interpreter lock (if it has been created and thread support is enabled) and reset the thread

state to NULL, returning the previous thread state (which is not NULL). If the lock has been created, the current thread must have acquired it.

void **PyEval_RestoreThread**(*PyThreadState* *tstate*)
Acquire the global interpreter lock (if it has been created and thread support is enabled) and set the thread state to *tstate*, which must not be NULL. If the lock has been created, the current thread must not have acquired it, otherwise deadlock ensues.

참고: Calling this function from a thread when the runtime is finalizing will terminate the thread, even if the thread was not created by Python. You can use `_Py_IsFinalizing()` or `sys.is_finalizing()` to check if the interpreter is in process of being finalized before calling this function to avoid unwanted termination.

---

*PyThreadState** **PyThreadState_Get**()
Return the current thread state. The global interpreter lock must be held. When the current thread state is NULL, this issues a fatal error (so that the caller needn't check for NULL).

*PyThreadState** **PyThreadState_Swap**(*PyThreadState* *tstate*)
Swap the current thread state with the thread state given by the argument *tstate*, which may be NULL. The global interpreter lock must be held and is not released.

void **PyEval_ReInitThreads**()
This function is called from *PyOS_AfterFork_Child()* to ensure that newly created child processes don't hold locks referring to threads which are not running in the child process.

The following functions use thread-local storage, and are not compatible with sub-interpreters:

PyGILState_STATE **PyGILState_Ensure**()
Ensure that the current thread is ready to call the Python C API regardless of the current state of Python, or of the global interpreter lock. This may be called as many times as desired by a thread as long as each call is matched with a call to *PyGILState_Release()*. In general, other thread-related APIs may be used between *PyGILState_Ensure()* and *PyGILState_Release()* calls as long as the thread state is restored to its previous state before the Release(). For example, normal usage of the *Py_BEGIN_ALLOW_THREADS* and *Py_END_ALLOW_THREADS* macros is acceptable.

The return value is an opaque "handle" to the thread state when *PyGILState_Ensure()* was called, and must be passed to *PyGILState_Release()* to ensure Python is left in the same state. Even though recursive calls are allowed, these handles *cannot* be shared - each unique call to *PyGILState_Ensure()* must save the handle for its call to *PyGILState_Release()*.

When the function returns, the current thread will hold the GIL and be able to call arbitrary Python code. Failure is a fatal error.

참고: Calling this function from a thread when the runtime is finalizing will terminate the thread, even if the thread was not created by Python. You can use `_Py_IsFinalizing()` or `sys.is_finalizing()` to check if the interpreter is in process of being finalized before calling this function to avoid unwanted termination.

---

void **PyGILState_Release**(PyGILState_STATE)
Release any resources previously acquired. After this call, Python's state will be the same as it was prior to the corresponding *PyGILState_Ensure()* call (but generally this state will be unknown to the caller, hence the use of the GILState API).

Every call to *PyGILState_Ensure()* must be matched by a call to *PyGILState_Release()* on the same thread.

*PyThreadState** **PyGILState_GetThisThreadState**()
Get the current thread state for this thread. May return NULL if no GILState API has been used on the current thread. Note that the main thread always has such a thread-state, even if no auto-thread-state call has been made on the main thread. This is mainly a helper/diagnostic function.

int **PyGILState_Check**()
Return 1 if the current thread is holding the GIL and 0 otherwise. This function can be called from any thread at any time. Only if it has had its Python thread state initialized and currently is holding the GIL will it return 1. This is mainly a helper/diagnostic function. It can be useful for example in callback contexts or memory allocation functions when knowing that the GIL is locked can allow the caller to perform sensitive actions or otherwise behave differently.

버전 3.4에 추가.

The following macros are normally used without a trailing semicolon; look for example usage in the Python source distribution.

Py_BEGIN_ALLOW_THREADS
This macro expands to `{ PyThreadState *_save; _save = PyEval_SaveThread();`. Note that it contains an opening brace; it must be matched with a following *Py_END_ALLOW_THREADS* macro. See above for further discussion of this macro.

Py_END_ALLOW_THREADS
This macro expands to `PyEval_RestoreThread(_save); }`. Note that it contains a closing brace; it must be matched with an earlier *Py_BEGIN_ALLOW_THREADS* macro. See above for further discussion of this macro.

Py_BLOCK_THREADS
This macro expands to `PyEval_RestoreThread(_save);`: it is equivalent to *Py_END_ALLOW_THREADS* without the closing brace.

Py_UNBLOCK_THREADS
This macro expands to `_save = PyEval_SaveThread();`: it is equivalent to *Py_BEGIN_ALLOW_THREADS* without the opening brace and variable declaration.

9.5.4 Low-level API

All of the following functions must be called after *Py_Initialize()*.

버전 3.7에서 변경: *Py_Initialize()* now initializes the *GIL*.

*PyInterpreterState** **PyInterpreterState_New**()
Create a new interpreter state object. The global interpreter lock need not be held, but may be held if it is necessary to serialize calls to this function.

void **PyInterpreterState_Clear**(*PyInterpreterState *interp*)
Reset all information in an interpreter state object. The global interpreter lock must be held.

void **PyInterpreterState_Delete**(*PyInterpreterState *interp*)
Destroy an interpreter state object. The global interpreter lock need not be held. The interpreter state must have been reset with a previous call to *PyInterpreterState_Clear()*.

*PyThreadState** **PyThreadState_New**(*PyInterpreterState *interp*)
Create a new thread state object belonging to the given interpreter object. The global interpreter lock need not be held, but may be held if it is necessary to serialize calls to this function.

void **PyThreadState_Clear**(*PyThreadState *tstate*)
Reset all information in a thread state object. The global interpreter lock must be held.

void **PyThreadState_Delete**(*PyThreadState *tstate*)
Destroy a thread state object. The global interpreter lock need not be held. The thread state must have been reset with a previous call to *PyThreadState_Clear()*.

PY_INT64_T **PyInterpreterState_GetID**(*PyInterpreterState *interp*)
Return the interpreter's unique ID. If there was any error in doing so then −1 is returned and an error is set.

버전 3.7에 추가.

*PyObject** **PyThreadState_GetDict**()
Return value: Borrowed reference. Return a dictionary in which extensions can store thread-specific state

information. Each extension should use a unique key to use to store state in the dictionary. It is okay to call this function when no current thread state is available. If this function returns NULL, no exception has been raised and the caller should assume no current thread state is available.

int **PyThreadState_SetAsyncExc**(unsigned long *id*, *PyObject* **exc*)
Asynchronously raise an exception in a thread. The *id* argument is the thread id of the target thread; *exc* is the exception object to be raised. This function does not steal any references to *exc*. To prevent naive misuse, you must write your own C extension to call this. Must be called with the GIL held. Returns the number of thread states modified; this is normally one, but will be zero if the thread id isn't found. If *exc* is NULL, the pending exception (if any) for the thread is cleared. This raises no exceptions.

버전 3.7에서 변경: The type of the *id* parameter changed from long to unsigned long.

void **PyEval_AcquireThread**(*PyThreadState* **tstate*)
Acquire the global interpreter lock and set the current thread state to *tstate*, which should not be NULL. The lock must have been created earlier. If this thread already has the lock, deadlock ensues.

PyEval_RestoreThread() is a higher-level function which is always available (even when threads have not been initialized).

void **PyEval_ReleaseThread**(*PyThreadState* **tstate*)
Reset the current thread state to NULL and release the global interpreter lock. The lock must have been created earlier and must be held by the current thread. The *tstate* argument, which must not be NULL, is only used to check that it represents the current thread state — if it isn't, a fatal error is reported.

PyEval_SaveThread() is a higher-level function which is always available (even when threads have not been initialized).

void **PyEval_AcquireLock**()
Acquire the global interpreter lock. The lock must have been created earlier. If this thread already has the lock, a deadlock ensues.

버전 3.2부터 폐지: This function does not update the current thread state. Please use *PyEval_RestoreThread()* or *PyEval_AcquireThread()* instead.

void **PyEval_ReleaseLock**()
Release the global interpreter lock. The lock must have been created earlier.

버전 3.2부터 폐지: This function does not update the current thread state. Please use *PyEval_SaveThread()* or *PyEval_ReleaseThread()* instead.

9.6 Sub-interpreter support

While in most uses, you will only embed a single Python interpreter, there are cases where you need to create several independent interpreters in the same process and perhaps even in the same thread. Sub-interpreters allow you to do that. You can switch between sub-interpreters using the *PyThreadState_Swap()* function. You can create and destroy them using the following functions:

*PyThreadState** **Py_NewInterpreter**()
Create a new sub-interpreter. This is an (almost) totally separate environment for the execution of Python code. In particular, the new interpreter has separate, independent versions of all imported modules, including the fundamental modules builtins, __main__ and sys. The table of loaded modules (sys.modules) and the module search path (sys.path) are also separate. The new environment has no sys.argv variable. It has new standard I/O stream file objects sys.stdin, sys.stdout and sys.stderr (however these refer to the same underlying file descriptors).

The return value points to the first thread state created in the new sub-interpreter. This thread state is made in the current thread state. Note that no actual thread is created; see the discussion of thread states below. If creation of the new interpreter is unsuccessful, NULL is returned; no exception is set since the exception state is stored in the current thread state and there may not be a current thread state. (Like all other Python/C API functions, the global interpreter lock must be held before calling this function and is still held when it returns; however, unlike most other Python/C API functions, there needn't be a current thread state on entry.)

Extension modules are shared between (sub-)interpreters as follows: the first time a particular extension is imported, it is initialized normally, and a (shallow) copy of its module's dictionary is squirreled away. When the same extension is imported by another (sub-)interpreter, a new module is initialized and filled with the contents of this copy; the extension's `init` function is not called. Note that this is different from what happens when an extension is imported after the interpreter has been completely re-initialized by calling *`Py_FinalizeEx()`* and *`Py_Initialize()`*; in that case, the extension's `initmodule` function *is* called again.

void **`Py_EndInterpreter`**(*PyThreadState* *tstate*)

Destroy the (sub-)interpreter represented by the given thread state. The given thread state must be the current thread state. See the discussion of thread states below. When the call returns, the current thread state is `NULL`. All thread states associated with this interpreter are destroyed. (The global interpreter lock must be held before calling this function and is still held when it returns.) *`Py_FinalizeEx()`* will destroy all sub-interpreters that haven't been explicitly destroyed at that point.

9.6.1 Bugs and caveats

Because sub-interpreters (and the main interpreter) are part of the same process, the insulation between them isn't perfect — for example, using low-level file operations like `os.close()` they can (accidentally or maliciously) affect each other's open files. Because of the way extensions are shared between (sub-)interpreters, some extensions may not work properly; this is especially likely when the extension makes use of (static) global variables, or when the extension manipulates its module's dictionary after its initialization. It is possible to insert objects created in one sub-interpreter into a namespace of another sub-interpreter; this should be done with great care to avoid sharing user-defined functions, methods, instances or classes between sub-interpreters, since import operations executed by such objects may affect the wrong (sub-)interpreter's dictionary of loaded modules.

Also note that combining this functionality with `PyGILState_*()` APIs is delicate, because these APIs assume a bijection between Python thread states and OS-level threads, an assumption broken by the presence of sub-interpreters. It is highly recommended that you don't switch sub-interpreters between a pair of matching *`PyGILState_Ensure()`* and *`PyGILState_Release()`* calls. Furthermore, extensions (such as `ctypes`) using these APIs to allow calling of Python code from non-Python created threads will probably be broken when using sub-interpreters.

9.7 Asynchronous Notifications

A mechanism is provided to make asynchronous notifications to the main interpreter thread. These notifications take the form of a function pointer and a void pointer argument.

int **`Py_AddPendingCall`**(int (**func*)(void *), void **arg*)

Schedule a function to be called from the main interpreter thread. On success, `0` is returned and *func* is queued for being called in the main thread. On failure, `-1` is returned without setting any exception.

When successfully queued, *func* will be *eventually* called from the main interpreter thread with the argument *arg*. It will be called asynchronously with respect to normally running Python code, but with both these conditions met:

- on a *bytecode* boundary;
- with the main thread holding the *global interpreter lock* (*func* can therefore use the full C API).

func must return `0` on success, or `-1` on failure with an exception set. *func* won't be interrupted to perform another asynchronous notification recursively, but it can still be interrupted to switch threads if the global interpreter lock is released.

This function doesn't need a current thread state to run, and it doesn't need the global interpreter lock.

> **경고:** This is a low-level function, only useful for very special cases. There is no guarantee that *func* will be called as quick as possible. If the main thread is busy executing a system call, *func* won't be called

before the system call returns. This function is generally **not** suitable for calling Python code from arbitrary C threads. Instead, use the *PyGILState API*.

버전 3.1에 추가.

9.8 Profiling and Tracing

The Python interpreter provides some low-level support for attaching profiling and execution tracing facilities. These are used for profiling, debugging, and coverage analysis tools.

This C interface allows the profiling or tracing code to avoid the overhead of calling through Python-level callable objects, making a direct C function call instead. The essential attributes of the facility have not changed; the interface allows trace functions to be installed per-thread, and the basic events reported to the trace function are the same as had been reported to the Python-level trace functions in previous versions.

int **(*Py_tracefunc)**(*PyObject* *obj, *PyFrameObject* *frame, int what, *PyObject* *arg)
The type of the trace function registered using `PyEval_SetProfile()` and `PyEval_SetTrace()`. The first parameter is the object passed to the registration function as *obj*, *frame* is the frame object to which the event pertains, *what* is one of the constants `PyTrace_CALL`, `PyTrace_EXCEPTION`, `PyTrace_LINE`, `PyTrace_RETURN`, `PyTrace_C_CALL`, `PyTrace_C_EXCEPTION`, `PyTrace_C_RETURN`, or `PyTrace_OPCODE`, and *arg* depends on the value of *what*:

Value of *what*	Meaning of *arg*
`PyTrace_CALL`	Always `Py_None`.
`PyTrace_EXCEPTION`	Exception information as returned by `sys.exc_info()`.
`PyTrace_LINE`	Always `Py_None`.
`PyTrace_RETURN`	Value being returned to the caller, or `NULL` if caused by an exception.
`PyTrace_C_CALL`	Function object being called.
`PyTrace_C_EXCEPTION`	Function object being called.
`PyTrace_C_RETURN`	Function object being called.
`PyTrace_OPCODE`	Always `Py_None`.

int **PyTrace_CALL**
The value of the *what* parameter to a `Py_tracefunc` function when a new call to a function or method is being reported, or a new entry into a generator. Note that the creation of the iterator for a generator function is not reported as there is no control transfer to the Python bytecode in the corresponding frame.

int **PyTrace_EXCEPTION**
The value of the *what* parameter to a `Py_tracefunc` function when an exception has been raised. The callback function is called with this value for *what* when after any bytecode is processed after which the exception becomes set within the frame being executed. The effect of this is that as exception propagation causes the Python stack to unwind, the callback is called upon return to each frame as the exception propagates. Only trace functions receives these events; they are not needed by the profiler.

int **PyTrace_LINE**
The value passed as the *what* parameter to a `Py_tracefunc` function (but not a profiling function) when a line-number event is being reported. It may be disabled for a frame by setting `f_trace_lines` to *0* on that frame.

int **PyTrace_RETURN**
The value for the *what* parameter to `Py_tracefunc` functions when a call is about to return.

int **PyTrace_C_CALL**
The value for the *what* parameter to `Py_tracefunc` functions when a C function is about to be called.

int **PyTrace_C_EXCEPTION**
The value for the *what* parameter to `Py_tracefunc` functions when a C function has raised an exception.

int **PyTrace_C_RETURN**
The value for the *what* parameter to `Py_tracefunc` functions when a C function has returned.

int **PyTrace_OPCODE**
The value for the *what* parameter to `Py_tracefunc` functions (but not profiling functions) when a new opcode is about to be executed. This event is not emitted by default: it must be explicitly requested by setting `f_trace_opcodes` to *1* on the frame.

void **PyEval_SetProfile**(*Py_tracefunc func*, *PyObject *obj*)
Set the profiler function to *func*. The *obj* parameter is passed to the function as its first parameter, and may be any Python object, or `NULL`. If the profile function needs to maintain state, using a different value for *obj* for each thread provides a convenient and thread-safe place to store it. The profile function is called for all monitored events except `PyTrace_LINE` `PyTrace_OPCODE` and `PyTrace_EXCEPTION`.

void **PyEval_SetTrace**(*Py_tracefunc func*, *PyObject *obj*)
Set the tracing function to *func*. This is similar to `PyEval_SetProfile()`, except the tracing function does receive line-number events and per-opcode events, but does not receive any event related to C function objects being called. Any trace function registered using `PyEval_SetTrace()` will not receive `PyTrace_C_CALL`, `PyTrace_C_EXCEPTION` or `PyTrace_C_RETURN` as a value for the *what* parameter.

9.9 Advanced Debugger Support

These functions are only intended to be used by advanced debugging tools.

*PyInterpreterState** **PyInterpreterState_Head**()
Return the interpreter state object at the head of the list of all such objects.

*PyInterpreterState** **PyInterpreterState_Main**()
Return the main interpreter state object.

*PyInterpreterState** **PyInterpreterState_Next**(*PyInterpreterState *interp*)
Return the next interpreter state object after *interp* from the list of all such objects.

PyThreadState * **PyInterpreterState_ThreadHead**(*PyInterpreterState *interp*)
Return the pointer to the first `PyThreadState` object in the list of threads associated with the interpreter *interp*.

*PyThreadState** **PyThreadState_Next**(*PyThreadState *tstate*)
Return the next thread state object after *tstate* from the list of all such objects belonging to the same `PyInterpreterState` object.

9.10 Thread Local Storage Support

The Python interpreter provides low-level support for thread-local storage (TLS) which wraps the underlying native TLS implementation to support the Python-level thread local storage API (`threading.local`). The CPython C level APIs are similar to those offered by pthreads and Windows: use a thread key and functions to associate a `void*` value per thread.

The GIL does *not* need to be held when calling these functions; they supply their own locking.

Note that `Python.h` does not include the declaration of the TLS APIs, you need to include `pythread.h` to use thread-local storage.

참고: None of these API functions handle memory management on behalf of the `void*` values. You need to allocate and deallocate them yourself. If the `void*` values happen to be `PyObject*`, these functions don't do refcount operations on them either.

9.10.1 Thread Specific Storage (TSS) API

TSS API is introduced to supersede the use of the existing TLS API within the CPython interpreter. This API uses a new type *Py_tss_t* instead of `int` to represent thread keys.

버전 3.7에 추가.

더 보기:

"A New C-API for Thread-Local Storage in CPython" (**PEP 539**)

Py_tss_t
This data structure represents the state of a thread key, the definition of which may depend on the underlying TLS implementation, and it has an internal field representing the key's initialization state. There are no public members in this structure.

When *Py_LIMITED_API* is not defined, static allocation of this type by *Py_tss_NEEDS_INIT* is allowed.

Py_tss_NEEDS_INIT
This macro expands to the initializer for *Py_tss_t* variables. Note that this macro won't be defined with *Py_LIMITED_API*.

Dynamic Allocation

Dynamic allocation of the *Py_tss_t*, required in extension modules built with *Py_LIMITED_API*, where static allocation of this type is not possible due to its implementation being opaque at build time.

*Py_tss_t** **PyThread_tss_alloc**()
Return a value which is the same state as a value initialized with *Py_tss_NEEDS_INIT*, or `NULL` in the case of dynamic allocation failure.

void **PyThread_tss_free**(*Py_tss_t *key*)
Free the given *key* allocated by *PyThread_tss_alloc()*, after first calling *PyThread_tss_delete()* to ensure any associated thread locals have been unassigned. This is a no-op if the *key* argument is *NULL*.

참고: A freed key becomes a dangling pointer, you should reset the key to *NULL*.

Methods

The parameter *key* of these functions must not be `NULL`. Moreover, the behaviors of *PyThread_tss_set()* and *PyThread_tss_get()* are undefined if the given *Py_tss_t* has not been initialized by *PyThread_tss_create()*.

int **PyThread_tss_is_created**(*Py_tss_t *key*)
Return a non-zero value if the given *Py_tss_t* has been initialized by *PyThread_tss_create()*.

int **PyThread_tss_create**(*Py_tss_t *key*)
Return a zero value on successful initialization of a TSS key. The behavior is undefined if the value pointed to by the *key* argument is not initialized by *Py_tss_NEEDS_INIT*. This function can be called repeatedly on the same key – calling it on an already initialized key is a no-op and immediately returns success.

void **PyThread_tss_delete**(*Py_tss_t *key*)
Destroy a TSS key to forget the values associated with the key across all threads, and change the key's initialization state to uninitialized. A destroyed key is able to be initialized again by *PyThread_tss_create()*. This function can be called repeatedly on the same key – calling it on an already destroyed key is a no-op.

int **PyThread_tss_set**(*Py_tss_t *key*, void *value*)
Return a zero value to indicate successfully associating a `void*` value with a TSS key in the current thread. Each thread has a distinct mapping of the key to a `void*` value.

void* **PyThread_tss_get**(*Py_tss_t* *key)
Return the `void*` value associated with a TSS key in the current thread. This returns `NULL` if no value is associated with the key in the current thread.

9.10.2 Thread Local Storage (TLS) API

버전 3.7부터 폐지: This API is superseded by *Thread Specific Storage (TSS) API*.

참고: This version of the API does not support platforms where the native TLS key is defined in a way that cannot be safely cast to `int`. On such platforms, *`PyThread_create_key()`* will return immediately with a failure status, and the other TLS functions will all be no-ops on such platforms.

Due to the compatibility problem noted above, this version of the API should not be used in new code.

int **PyThread_create_key**()

void **PyThread_delete_key**(int *key*)

int **PyThread_set_key_value**(int *key*, void **value*)

void* **PyThread_get_key_value**(int *key*)

void **PyThread_delete_key_value**(int *key*)

void **PyThread_ReInitTLS**()

CHAPTER 10

Memory Management

10.1 Overview

Memory management in Python involves a private heap containing all Python objects and data structures. The management of this private heap is ensured internally by the *Python memory manager*. The Python memory manager has different components which deal with various dynamic storage management aspects, like sharing, segmentation, preallocation or caching.

At the lowest level, a raw memory allocator ensures that there is enough room in the private heap for storing all Python-related data by interacting with the memory manager of the operating system. On top of the raw memory allocator, several object-specific allocators operate on the same heap and implement distinct memory management policies adapted to the peculiarities of every object type. For example, integer objects are managed differently within the heap than strings, tuples or dictionaries because integers imply different storage requirements and speed/space tradeoffs. The Python memory manager thus delegates some of the work to the object-specific allocators, but ensures that the latter operate within the bounds of the private heap.

It is important to understand that the management of the Python heap is performed by the interpreter itself and that the user has no control over it, even if they regularly manipulate object pointers to memory blocks inside that heap. The allocation of heap space for Python objects and other internal buffers is performed on demand by the Python memory manager through the Python/C API functions listed in this document.

To avoid memory corruption, extension writers should never try to operate on Python objects with the functions exported by the C library: `malloc()`, `calloc()`, `realloc()` and `free()`. This will result in mixed calls between the C allocator and the Python memory manager with fatal consequences, because they implement different algorithms and operate on different heaps. However, one may safely allocate and release memory blocks with the C library allocator for individual purposes, as shown in the following example:

```
PyObject *res;
char *buf = (char *) malloc(BUFSIZ); /* for I/O */

if (buf == NULL)
    return PyErr_NoMemory();
...Do some I/O operation involving buf...
res = PyBytes_FromString(buf);
free(buf); /* malloc'ed */
return res;
```

In this example, the memory request for the I/O buffer is handled by the C library allocator. The Python memory manager is involved only in the allocation of the bytes object returned as a result.

In most situations, however, it is recommended to allocate memory from the Python heap specifically because the latter is under control of the Python memory manager. For example, this is required when the interpreter is extended with new object types written in C. Another reason for using the Python heap is the desire to *inform* the Python memory manager about the memory needs of the extension module. Even when the requested memory is used exclusively for internal, highly-specific purposes, delegating all memory requests to the Python memory manager causes the interpreter to have a more accurate image of its memory footprint as a whole. Consequently, under certain circumstances, the Python memory manager may or may not trigger appropriate actions, like garbage collection, memory compaction or other preventive procedures. Note that by using the C library allocator as shown in the previous example, the allocated memory for the I/O buffer escapes completely the Python memory manager.

더 보기:

The `PYTHONMALLOC` environment variable can be used to configure the memory allocators used by Python.

The `PYTHONMALLOCSTATS` environment variable can be used to print statistics of the *pymalloc memory allocator* every time a new pymalloc object arena is created, and on shutdown.

10.2 Raw Memory Interface

The following function sets are wrappers to the system allocator. These functions are thread-safe, the *GIL* does not need to be held.

The *default raw memory allocator* uses the following functions: `malloc()`, `calloc()`, `realloc()` and `free()`; call `malloc(1)` (or `calloc(1, 1)`) when requesting zero bytes.

버전 3.4에 추가.

void* **PyMem_RawMalloc**(size_t *n*)
Allocates *n* bytes and returns a pointer of type `void*` to the allocated memory, or `NULL` if the request fails.

Requesting zero bytes returns a distinct non-`NULL` pointer if possible, as if `PyMem_RawMalloc(1)` had been called instead. The memory will not have been initialized in any way.

void* **PyMem_RawCalloc**(size_t *nelem*, size_t *elsize*)
Allocates *nelem* elements each whose size in bytes is *elsize* and returns a pointer of type `void*` to the allocated memory, or `NULL` if the request fails. The memory is initialized to zeros.

Requesting zero elements or elements of size zero bytes returns a distinct non-`NULL` pointer if possible, as if `PyMem_RawCalloc(1, 1)` had been called instead.

버전 3.5에 추가.

void* **PyMem_RawRealloc**(void **p*, size_t *n*)
Resizes the memory block pointed to by *p* to *n* bytes. The contents will be unchanged to the minimum of the old and the new sizes.

If *p* is `NULL`, the call is equivalent to `PyMem_RawMalloc(n)`; else if *n* is equal to zero, the memory block is resized but is not freed, and the returned pointer is non-`NULL`.

Unless *p* is `NULL`, it must have been returned by a previous call to *PyMem_RawMalloc()*, *PyMem_RawRealloc()* or *PyMem_RawCalloc()*.

If the request fails, *PyMem_RawRealloc()* returns `NULL` and *p* remains a valid pointer to the previous memory area.

void **PyMem_RawFree**(void **p*)
Frees the memory block pointed to by *p*, which must have been returned by a previous call to *PyMem_RawMalloc()*, *PyMem_RawRealloc()* or *PyMem_RawCalloc()*. Otherwise, or if `PyMem_RawFree(p)` has been called before, undefined behavior occurs.

If *p* is `NULL`, no operation is performed.

10.3 Memory Interface

The following function sets, modeled after the ANSI C standard, but specifying behavior when requesting zero bytes, are available for allocating and releasing memory from the Python heap.

The *default memory allocator* uses the *pymalloc memory allocator*.

> **경고:** The *GIL* must be held when using these functions.

버전 3.6에서 변경: The default allocator is now pymalloc instead of system `malloc()`.

void* **PyMem_Malloc**(size_t *n*)
Allocates *n* bytes and returns a pointer of type `void*` to the allocated memory, or `NULL` if the request fails.

Requesting zero bytes returns a distinct non-`NULL` pointer if possible, as if `PyMem_Malloc(1)` had been called instead. The memory will not have been initialized in any way.

void* **PyMem_Calloc**(size_t *nelem*, size_t *elsize*)
Allocates *nelem* elements each whose size in bytes is *elsize* and returns a pointer of type `void*` to the allocated memory, or `NULL` if the request fails. The memory is initialized to zeros.

Requesting zero elements or elements of size zero bytes returns a distinct non-`NULL` pointer if possible, as if `PyMem_Calloc(1, 1)` had been called instead.

버전 3.5에 추가.

void* **PyMem_Realloc**(void **p*, size_t *n*)
Resizes the memory block pointed to by *p* to *n* bytes. The contents will be unchanged to the minimum of the old and the new sizes.

If *p* is `NULL`, the call is equivalent to `PyMem_Malloc(n)`; else if *n* is equal to zero, the memory block is resized but is not freed, and the returned pointer is non-`NULL`.

Unless *p* is `NULL`, it must have been returned by a previous call to *`PyMem_Malloc()`*, *`PyMem_Realloc()`* or *`PyMem_Calloc()`*.

If the request fails, *`PyMem_Realloc()`* returns `NULL` and *p* remains a valid pointer to the previous memory area.

void **PyMem_Free**(void **p*)
Frees the memory block pointed to by *p*, which must have been returned by a previous call to *`PyMem_Malloc()`*, *`PyMem_Realloc()`* or *`PyMem_Calloc()`*. Otherwise, or if `PyMem_Free(p)` has been called before, undefined behavior occurs.

If *p* is `NULL`, no operation is performed.

The following type-oriented macros are provided for convenience. Note that *TYPE* refers to any C type.

TYPE* **PyMem_New**(TYPE, size_t *n*)
Same as *`PyMem_Malloc()`*, but allocates `(n * sizeof(TYPE))` bytes of memory. Returns a pointer cast to `TYPE*`. The memory will not have been initialized in any way.

TYPE* **PyMem_Resize**(void **p*, TYPE, size_t *n*)
Same as *`PyMem_Realloc()`*, but the memory block is resized to `(n * sizeof(TYPE))` bytes. Returns a pointer cast to `TYPE*`. On return, *p* will be a pointer to the new memory area, or `NULL` in the event of failure.

This is a C preprocessor macro; *p* is always reassigned. Save the original value of *p* to avoid losing memory when handling errors.

void **PyMem_Del**(void **p*)
Same as *`PyMem_Free()`*.

In addition, the following macro sets are provided for calling the Python memory allocator directly, without involving the C API functions listed above. However, note that their use does not preserve binary compatibility across Python versions and is therefore deprecated in extension modules.

- `PyMem_MALLOC(size)`
- `PyMem_NEW(type, size)`
- `PyMem_REALLOC(ptr, size)`
- `PyMem_RESIZE(ptr, type, size)`
- `PyMem_FREE(ptr)`
- `PyMem_DEL(ptr)`

10.4 Object allocators

The following function sets, modeled after the ANSI C standard, but specifying behavior when requesting zero bytes, are available for allocating and releasing memory from the Python heap.

The *default object allocator* uses the *pymalloc memory allocator*.

> **경고:** The *GIL* must be held when using these functions.

void* **PyObject_Malloc**(size_t *n*)
Allocates *n* bytes and returns a pointer of type `void*` to the allocated memory, or `NULL` if the request fails.

Requesting zero bytes returns a distinct non-`NULL` pointer if possible, as if `PyObject_Malloc(1)` had been called instead. The memory will not have been initialized in any way.

void* **PyObject_Calloc**(size_t *nelem*, size_t *elsize*)
Allocates *nelem* elements each whose size in bytes is *elsize* and returns a pointer of type `void*` to the allocated memory, or `NULL` if the request fails. The memory is initialized to zeros.

Requesting zero elements or elements of size zero bytes returns a distinct non-`NULL` pointer if possible, as if `PyObject_Calloc(1, 1)` had been called instead.

버전 3.5에 추가.

void* **PyObject_Realloc**(void **p*, size_t *n*)
Resizes the memory block pointed to by *p* to *n* bytes. The contents will be unchanged to the minimum of the old and the new sizes.

If *p* is `NULL`, the call is equivalent to `PyObject_Malloc(n)`; else if *n* is equal to zero, the memory block is resized but is not freed, and the returned pointer is non-`NULL`.

Unless *p* is `NULL`, it must have been returned by a previous call to *PyObject_Malloc()*, *PyObject_Realloc()* or *PyObject_Calloc()*.

If the request fails, *PyObject_Realloc()* returns `NULL` and *p* remains a valid pointer to the previous memory area.

void **PyObject_Free**(void **p*)
Frees the memory block pointed to by *p*, which must have been returned by a previous call to *PyObject_Malloc()*, *PyObject_Realloc()* or *PyObject_Calloc()*. Otherwise, or if `PyObject_Free(p)` has been called before, undefined behavior occurs.

If *p* is `NULL`, no operation is performed.

10.5 Default Memory Allocators

Default memory allocators:

Configuration	Name	PyMem_RawMalloc	PyMem_Malloc	PyObject_Malloc
Release build	`"pymalloc"`	`malloc`	`pymalloc`	`pymalloc`
Debug build	`"pymalloc_debug"`	`malloc` + debug	`pymalloc` + debug	`pymalloc` + debug
Release build, without pymalloc	`"malloc"`	`malloc`	`malloc`	`malloc`
Debug build, without pymalloc	`"malloc_debug"`	`malloc` + debug	`malloc` + debug	`malloc` + debug

Legend:

- Name: value for `PYTHONMALLOC` environment variable
- `malloc`: system allocators from the standard C library, C functions: `malloc()`, `calloc()`, `realloc()` and `free()`
- `pymalloc`: *pymalloc memory allocator*
- "+ debug": with debug hooks installed by *`PyMem_SetupDebugHooks()`*

10.6 Customize Memory Allocators

버전 3.4에 추가.

`PyMemAllocatorEx`
Structure used to describe a memory block allocator. The structure has four fields:

Field	Meaning
`void *ctx`	user context passed as first argument
`void* malloc(void *ctx, size_t size)`	allocate a memory block
`void* calloc(void *ctx, size_t nelem, size_t elsize)`	allocate a memory block initialized with zeros
`void* realloc(void *ctx, void *ptr, size_t new_size)`	allocate or resize a memory block
`void free(void *ctx, void *ptr)`	free a memory block

버전 3.5에서 변경: The `PyMemAllocator` structure was renamed to *`PyMemAllocatorEx`* and a new `calloc` field was added.

`PyMemAllocatorDomain`
Enum used to identify an allocator domain. Domains:

`PYMEM_DOMAIN_RAW`
Functions:

- *`PyMem_RawMalloc()`*
- *`PyMem_RawRealloc()`*
- *`PyMem_RawCalloc()`*
- *`PyMem_RawFree()`*

`PYMEM_DOMAIN_MEM`
Functions:

- *PyMem_Malloc()*,
- *PyMem_Realloc()*
- *PyMem_Calloc()*
- *PyMem_Free()*

PYMEM_DOMAIN_OBJ
Functions:

- *PyObject_Malloc()*
- *PyObject_Realloc()*
- *PyObject_Calloc()*
- *PyObject_Free()*

void **PyMem_GetAllocator**(*PyMemAllocatorDomain domain*, *PyMemAllocatorEx *allocator*)
Get the memory block allocator of the specified domain.

void **PyMem_SetAllocator**(*PyMemAllocatorDomain domain*, *PyMemAllocatorEx *allocator*)
Set the memory block allocator of the specified domain.

The new allocator must return a distinct non-NULL pointer when requesting zero bytes.

For the *PYMEM_DOMAIN_RAW* domain, the allocator must be thread-safe: the *GIL* is not held when the allocator is called.

If the new allocator is not a hook (does not call the previous allocator), the *PyMem_SetupDebugHooks()* function must be called to reinstall the debug hooks on top on the new allocator.

void **PyMem_SetupDebugHooks**(void)
Setup hooks to detect bugs in the Python memory allocator functions.

Newly allocated memory is filled with the byte 0xCD (CLEANBYTE), freed memory is filled with the byte 0xDD (DEADBYTE). Memory blocks are surrounded by "forbidden bytes" (FORBIDDENBYTE: byte 0xFD).

Runtime checks:

- Detect API violations, ex: *PyObject_Free()* called on a buffer allocated by *PyMem_Malloc()*
- Detect write before the start of the buffer (buffer underflow)
- Detect write after the end of the buffer (buffer overflow)
- Check that the *GIL* is held when allocator functions of *PYMEM_DOMAIN_OBJ* (ex: *PyObject_Malloc()*) and *PYMEM_DOMAIN_MEM* (ex: *PyMem_Malloc()*) domains are called

On error, the debug hooks use the tracemalloc module to get the traceback where a memory block was allocated. The traceback is only displayed if tracemalloc is tracing Python memory allocations and the memory block was traced.

These hooks are *installed by default* if Python is compiled in debug mode. The PYTHONMALLOC environment variable can be used to install debug hooks on a Python compiled in release mode.

버전 3.6에서 변경: This function now also works on Python compiled in release mode. On error, the debug hooks now use tracemalloc to get the traceback where a memory block was allocated. The debug hooks now also check if the GIL is held when functions of *PYMEM_DOMAIN_OBJ* and *PYMEM_DOMAIN_MEM* domains are called.

버전 3.7.3에서 변경: Byte patterns 0xCB (CLEANBYTE), 0xDB (DEADBYTE) and 0xFB (FORBIDDENBYTE) have been replaced with 0xCD, 0xDD and 0xFD to use the same values than Windows CRT debug malloc() and free().

10.7 The pymalloc allocator

Python has a *pymalloc* allocator optimized for small objects (smaller or equal to 512 bytes) with a short lifetime. It uses memory mappings called "arenas" with a fixed size of 256 KiB. It falls back to `PyMem_RawMalloc()` and `PyMem_RawRealloc()` for allocations larger than 512 bytes.

pymalloc is the *default allocator* of the `PYMEM_DOMAIN_MEM` (ex: `PyMem_Malloc()`) and `PYMEM_DOMAIN_OBJ` (ex: `PyObject_Malloc()`) domains.

The arena allocator uses the following functions:

- `VirtualAlloc()` and `VirtualFree()` on Windows,
- `mmap()` and `munmap()` if available,
- `malloc()` and `free()` otherwise.

10.7.1 Customize pymalloc Arena Allocator

버전 3.4에 추가.

PyObjectArenaAllocator
Structure used to describe an arena allocator. The structure has three fields:

Field	Meaning
`void *ctx`	user context passed as first argument
`void* alloc(void *ctx, size_t size)`	allocate an arena of size bytes
`void free(void *ctx, size_t size, void *ptr)`	free an arena

PyObject_GetArenaAllocator(*PyObjectArenaAllocator *allocator*)
Get the arena allocator.

PyObject_SetArenaAllocator(*PyObjectArenaAllocator *allocator*)
Set the arena allocator.

10.8 tracemalloc C API

버전 3.7에 추가.

int **PyTraceMalloc_Track**(unsigned int *domain*, uintptr_t *ptr*, size_t *size*)
Track an allocated memory block in the `tracemalloc` module.

Return `0` on success, return `-1` on error (failed to allocate memory to store the trace). Return `-2` if tracemalloc is disabled.

If memory block is already tracked, update the existing trace.

int **PyTraceMalloc_Untrack**(unsigned int *domain*, uintptr_t *ptr*)
Untrack an allocated memory block in the `tracemalloc` module. Do nothing if the block was not tracked.

Return `-2` if tracemalloc is disabled, otherwise return `0`.

10.9 Examples

Here is the example from section *Overview*, rewritten so that the I/O buffer is allocated from the Python heap by using the first function set:

```
PyObject *res;
char *buf = (char *) PyMem_Malloc(BUFSIZ); /* for I/O */

if (buf == NULL)
    return PyErr_NoMemory();
/* ...Do some I/O operation involving buf... */
res = PyBytes_FromString(buf);
PyMem_Free(buf); /* allocated with PyMem_Malloc */
return res;
```

The same code using the type-oriented function set:

```
PyObject *res;
char *buf = PyMem_New(char, BUFSIZ); /* for I/O */

if (buf == NULL)
    return PyErr_NoMemory();
/* ...Do some I/O operation involving buf... */
res = PyBytes_FromString(buf);
PyMem_Del(buf); /* allocated with PyMem_New */
return res;
```

Note that in the two examples above, the buffer is always manipulated via functions belonging to the same set. Indeed, it is required to use the same memory API family for a given memory block, so that the risk of mixing different allocators is reduced to a minimum. The following code sequence contains two errors, one of which is labeled as *fatal* because it mixes two different allocators operating on different heaps.

```
char *buf1 = PyMem_New(char, BUFSIZ);
char *buf2 = (char *) malloc(BUFSIZ);
char *buf3 = (char *) PyMem_Malloc(BUFSIZ);
...
PyMem_Del(buf3);  /* Wrong -- should be PyMem_Free() */
free(buf2);       /* Right -- allocated via malloc() */
free(buf1);       /* Fatal -- should be PyMem_Del()  */
```

In addition to the functions aimed at handling raw memory blocks from the Python heap, objects in Python are allocated and released with `PyObject_New()`, `PyObject_NewVar()` and `PyObject_Del()`.

These will be explained in the next chapter on defining and implementing new object types in C.

CHAPTER 11

객체 구현 지원

이 장에서는 새 객체 형을 정의할 때 사용되는 함수, 형 및 매크로에 관해 설명합니다.

11.1 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 None으로 노출되는 객체. 이 객체에 대한 포인터로 평가되는 *Py_None* 매크로를 사용해서 액세스해야 합니다.

더 보기:

PyModule_Create() 확장 모듈을 할당하고 만듭니다.

11.2 Common Object Structures

There are a large number of structures which are used in the definition of object types for Python. This section describes these structures and how they are used.

All Python objects ultimately share a small number of fields at the beginning of the object's representation in memory. These are represented by the *PyObject* and *PyVarObject* types, which are defined, in turn, by the expansions of some macros also used, whether directly or indirectly, in the definition of all other Python objects.

PyObject
All object types are extensions of this type. This is a type which contains the information Python needs to treat a pointer to an object as an object. In a normal "release" build, it contains only the object's reference count and a pointer to the corresponding type object. Nothing is actually declared to be a *PyObject*, but every pointer to a Python object can be cast to a *PyObject**. Access to the members must be done by using the macros *Py_REFCNT* and *Py_TYPE*.

PyVarObject
This is an extension of *PyObject* that adds the ob_size field. This is only used for objects that have some notion of *length*. This type does not often appear in the Python/C API. Access to the members must be done by using the macros *Py_REFCNT*, *Py_TYPE*, and *Py_SIZE*.

PyObject_HEAD
This is a macro used when declaring new types which represent objects without a varying length. The PyObject_HEAD macro expands to:

```
PyObject ob_base;
```

See documentation of *PyObject* above.

PyObject_VAR_HEAD
This is a macro used when declaring new types which represent objects with a length that varies from instance to instance. The PyObject_VAR_HEAD macro expands to:

```
PyVarObject ob_base;
```

See documentation of *PyVarObject* above.

Py_TYPE(o)
This macro is used to access the ob_type member of a Python object. It expands to:

```
(((PyObject*)(o))->ob_type)
```

Py_REFCNT(o)
This macro is used to access the ob_refcnt member of a Python object. It expands to:

```
(((PyObject*)(o))->ob_refcnt)
```

Py_SIZE(o)
This macro is used to access the ob_size member of a Python object. It expands to:

```
(((PyVarObject*)(o))->ob_size)
```

PyObject_HEAD_INIT(type)
This is a macro which expands to initialization values for a new *PyObject* type. This macro expands to:

```
_PyObject_EXTRA_INIT
1, type,
```

PyVarObject_HEAD_INIT(type, size)
This is a macro which expands to initialization values for a new *PyVarObject* type, including the ob_size field. This macro expands to:

```
_PyObject_EXTRA_INIT
1, type, size,
```

PyCFunction
Type of the functions used to implement most Python callables in C. Functions of this type take two *PyObject** parameters and return one such value. If the return value is NULL, an exception shall have been set. If not NULL, the return value is interpreted as the return value of the function as exposed in Python. The function must return a new reference.

PyCFunctionWithKeywords
Type of the functions used to implement Python callables in C with signature METH_VARARGS | METH_KEYWORDS.

_PyCFunctionFast
Type of the functions used to implement Python callables in C with signature *METH_FASTCALL*.

_PyCFunctionFastWithKeywords
Type of the functions used to implement Python callables in C with signature METH_FASTCALL | METH_KEYWORDS.

PyMethodDef
Structure used to describe a method of an extension type. This structure has four fields:

Field	C Type	Meaning
ml_name	const char *	name of the method
ml_meth	PyCFunction	pointer to the C implementation
ml_flags	int	flag bits indicating how the call should be constructed
ml_doc	const char *	points to the contents of the docstring

The ml_meth is a C function pointer. The functions may be of different types, but they always return *PyObject**. If the function is not of the *PyCFunction*, the compiler will require a cast in the method table. Even though *PyCFunction* defines the first parameter as *PyObject**, it is common that the method implementation uses the specific C type of the *self* object.

The ml_flags field is a bitfield which can include the following flags. The individual flags indicate either a calling convention or a binding convention.

There are four basic calling conventions for positional arguments and two of them can be combined with METH_KEYWORDS to support also keyword arguments. So there are a total of 6 calling conventions:

METH_VARARGS
This is the typical calling convention, where the methods have the type *PyCFunction*. The function expects two *PyObject** values. The first one is the *self* object for methods; for module functions, it is the module object. The second parameter (often called *args*) is a tuple object representing all arguments. This parameter is typically processed using *PyArg_ParseTuple()* or *PyArg_UnpackTuple()*.

METH_VARARGS | METH_KEYWORDS
Methods with these flags must be of type *PyCFunctionWithKeywords*. The function expects three parameters: *self*, *args*, *kwargs* where *kwargs* is a dictionary of all the keyword arguments or possibly NULL if there are no keyword arguments. The parameters are typically processed using *PyArg_ParseTupleAndKeywords()*.

METH_FASTCALL
Fast calling convention supporting only positional arguments. The methods have the type

_PyCFunctionFast. The first parameter is *self*, the second parameter is a C array of *PyObject** values indicating the arguments and the third parameter is the number of arguments (the length of the array).

This is not part of the *limited API*.

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 vectorcall 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.

This is not part of the *limited API*.

버전 3.7에 추가.

METH_NOARGS
Methods without parameters don't need to check whether arguments are given if they are listed with the *METH_NOARGS* flag. They need to be of type *PyCFunction*. The first parameter is typically named *self* and will hold a reference to the module or object instance. In all cases the second parameter will be `NULL`.

METH_O
Methods with a single object argument can be listed with the *METH_O* flag, instead of invoking *PyArg_ParseTuple()* with a `"O"` argument. They have the type *PyCFunction*, with the *self* parameter, and a *PyObject** parameter representing the single argument.

These two constants are not used to indicate the calling convention but the binding when use with methods of classes. These may not be used for functions defined for modules. At most one of these flags may be set for any given method.

METH_CLASS
The method will be passed the type object as the first parameter rather than an instance of the type. This is used to create *class methods*, similar to what is created when using the `classmethod()` built-in function.

METH_STATIC
The method will be passed `NULL` as the first parameter rather than an instance of the type. This is used to create *static methods*, similar to what is created when using the `staticmethod()` built-in function.

One other constant controls whether a method is loaded in place of another definition with the same method name.

METH_COEXIST
The method will be loaded in place of existing definitions. Without *METH_COEXIST*, the default is to skip repeated definitions. Since slot wrappers are loaded before the method table, the existence of a *sq_contains* slot, for example, would generate a wrapped method named `__contains__()` and preclude the loading of a corresponding PyCFunction with the same name. With the flag defined, the PyCFunction will be loaded in place of the wrapper object and will co-exist with the slot. This is helpful because calls to PyCFunctions are optimized more than wrapper object calls.

PyMemberDef
Structure which describes an attribute of a type which corresponds to a C struct member. Its fields are:

Field	C Type	Meaning
`name`	const char *	name of the member
`type`	int	the type of the member in the C struct
`offset`	Py_ssize_t	the offset in bytes that the member is located on the type's object struct
`flags`	int	flag bits indicating if the field should be read-only or writable
`doc`	const char *	points to the contents of the docstring

`type` can be one of many `T_` macros corresponding to various C types. When the member is accessed in Python, it will be converted to the equivalent Python type.

Macro name	C type
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

`T_OBJECT` and `T_OBJECT_EX` differ in that `T_OBJECT` returns `None` if the member is `NULL` and `T_OBJECT_EX` raises an `AttributeError`. Try to use `T_OBJECT_EX` over `T_OBJECT` because `T_OBJECT_EX` handles use of the `del` statement on that attribute more correctly than `T_OBJECT`.

`flags` can be `0` for write and read access or `READONLY` for read-only access. Using `T_STRING` for `type` implies `READONLY`. `T_STRING` data is interpreted as UTF-8. Only `T_OBJECT` and `T_OBJECT_EX` members can be deleted. (They are set to `NULL`).

PyGetSetDef

Structure to define property-like access for a type. See also description of the *PyTypeObject.tp_getset* slot.

Field	C Type	Meaning
name	const char *	attribute name
get	getter	C Function to get the attribute
set	setter	optional C function to set or delete the attribute, if omitted the attribute is readonly
doc	const char *	optional docstring
closure	void *	optional function pointer, providing additional data for getter and setter

The `get` function takes one *PyObject** parameter (the instance) and a function pointer (the associated `closure`):

```
typedef PyObject *(*getter)(PyObject *, void *);
```

It should return a new reference on success or `NULL` with a set exception on failure.

`set` functions take two *PyObject** parameters (the instance and the value to be set) and a function pointer (the associated `closure`):

```
typedef int (*setter)(PyObject *, PyObject *, void *);
```

In case the attribute should be deleted the second parameter is `NULL`. Should return `0` on success or `-1` with a set exception on failure.

11.3 Type Objects

Perhaps one of the most important structures of the Python object system is the structure that defines a new type: the *PyTypeObject* structure. Type objects can be handled using any of the `PyObject_*()` or `PyType_*()` functions, but do not offer much that's interesting to most Python applications. These objects are fundamental to how objects behave, so they are very important to the interpreter itself and to any extension module that implements new types.

Type objects are fairly large compared to most of the standard types. The reason for the size is that each type object stores a large number of values, mostly C function pointers, each of which implements a small part of the type's functionality. The fields of the type object are examined in detail in this section. The fields will be described in the order in which they occur in the structure.

Typedefs: unaryfunc, binaryfunc, ternaryfunc, inquiry, intargfunc, intintargfunc, intobjargproc, intintobjargproc, objobjargproc, destructor, freefunc, printfunc, getattrfunc, getattrofunc, setattrfunc, setattrofunc, reprfunc, hashfunc

The structure definition for *PyTypeObject* can be found in `Include/object.h`. For convenience of reference, this repeats the definition found there:

```
typedef struct _typeobject {
    PyObject_VAR_HEAD
    const char *tp_name; /* For printing, in format "<module>.<name>" */
    Py_ssize_t tp_basicsize, tp_itemsize; /* For allocation */

    /* Methods to implement standard operations */

    destructor tp_dealloc;
    printfunc tp_print;
    getattrfunc tp_getattr;
    setattrfunc tp_setattr;
    PyAsyncMethods *tp_as_async; /* formerly known as tp_compare (Python 2)
                                    or tp_reserved (Python 3) */
    reprfunc tp_repr;

    /* Method suites for standard classes */

    PyNumberMethods *tp_as_number;
    PySequenceMethods *tp_as_sequence;
    PyMappingMethods *tp_as_mapping;

    /* More standard operations (here for binary compatibility) */

    hashfunc tp_hash;
    ternaryfunc tp_call;
    reprfunc tp_str;
    getattrofunc tp_getattro;
    setattrofunc tp_setattro;

    /* Functions to access object as input/output buffer */
    PyBufferProcs *tp_as_buffer;

    /* Flags to define presence of optional/expanded features */
    unsigned long tp_flags;

    const char *tp_doc; /* Documentation string */

    /* call function for all accessible objects */
    traverseproc tp_traverse;

    /* delete references to contained objects */
    inquiry tp_clear;
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    /* rich comparisons */
    richcmpfunc tp_richcompare;

    /* weak reference enabler */
    Py_ssize_t tp_weaklistoffset;

    /* Iterators */
    getiterfunc tp_iter;
    iternextfunc tp_iternext;

    /* Attribute descriptor and subclassing stuff */
    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;
```

The type object structure extends the *PyVarObject* structure. The `ob_size` field is used for dynamic types (created by `type_new()`, usually called from a class statement). Note that *PyType_Type* (the metatype) initializes *tp_itemsize*, which means that its instances (i.e. type objects) *must* have the `ob_size` field.

*PyObject** **PyObject._ob_next**
*PyObject** **PyObject._ob_prev**
These fields are only present when the macro `Py_TRACE_REFS` is defined. Their initialization to `NULL` is taken care of by the `PyObject_HEAD_INIT` macro. For statically allocated objects, these fields always remain `NULL`. For dynamically allocated objects, these two fields are used to link the object into a doubly-linked list of *all* live objects on the heap. This could be used for various debugging purposes; currently the only use is to print the objects that are still alive at the end of a run when the environment variable `PYTHONDUMPREFS` is set.

These fields are not inherited by subtypes.

Py_ssize_t **PyObject.ob_refcnt**
This is the type object's reference count, initialized to `1` by the `PyObject_HEAD_INIT` macro. Note that for statically allocated type objects, the type's instances (objects whose `ob_type` points back to the type) do *not* count as references. But for dynamically allocated type objects, the instances *do* count as references.

This field is not inherited by subtypes.

*PyTypeObject** **PyObject.ob_type**
This is the type's type, in other words its metatype. It is initialized by the argument to the

PyObject_HEAD_INIT macro, and its value should normally be &PyType_Type. However, for dynamically loadable extension modules that must be usable on Windows (at least), the compiler complains that this is not a valid initializer. Therefore, the convention is to pass NULL to the PyObject_HEAD_INIT macro and to initialize this field explicitly at the start of the module's initialization function, before doing anything else. This is typically done like this:

```
Foo_Type.ob_type = &PyType_Type;
```

This should be done before any instances of the type are created. *PyType_Ready()* checks if ob_type is NULL, and if so, initializes it to the ob_type field of the base class. *PyType_Ready()* will not change this field if it is non-zero.

This field is inherited by subtypes.

Py_ssize_t **PyVarObject.ob_size**
For statically allocated type objects, this should be initialized to zero. For dynamically allocated type objects, this field has a special internal meaning.

This field is not inherited by subtypes.

const char* **PyTypeObject.tp_name**
Pointer to a NUL-terminated string containing the name of the type. For types that are accessible as module globals, the string should be the full module name, followed by a dot, followed by the type name; for built-in types, it should be just the type name. If the module is a submodule of a package, the full package name is part of the full module name. For example, a type named T defined in module M in subpackage Q in package P should have the *tp_name* initializer "P.Q.M.T".

For dynamically allocated type objects, this should just be the type name, and the module name explicitly stored in the type dict as the value for key '__module__'.

For statically allocated type objects, the tp_name field should contain a dot. Everything before the last dot is made accessible as the __module__ attribute, and everything after the last dot is made accessible as the __name__ attribute.

If no dot is present, the entire *tp_name* field is made accessible as the __name__ attribute, and the __module__ attribute is undefined (unless explicitly set in the dictionary, as explained above). This means your type will be impossible to pickle. Additionally, it will not be listed in module documentations created with pydoc.

This field is not inherited by subtypes.

Py_ssize_t **PyTypeObject.tp_basicsize**
Py_ssize_t **PyTypeObject.tp_itemsize**
These fields allow calculating the size in bytes of instances of the type.

There are two kinds of types: types with fixed-length instances have a zero *tp_itemsize* field, types with variable-length instances have a non-zero *tp_itemsize* field. For a type with fixed-length instances, all instances have the same size, given in *tp_basicsize*.

For a type with variable-length instances, the instances must have an ob_size field, and the instance size is *tp_basicsize* plus N times *tp_itemsize*, where N is the "length" of the object. The value of N is typically stored in the instance's ob_size field. There are exceptions: for example, ints use a negative ob_size to indicate a negative number, and N is abs(ob_size) there. Also, the presence of an ob_size field in the instance layout doesn't mean that the instance structure is variable-length (for example, the structure for the list type has fixed-length instances, yet those instances have a meaningful ob_size field).

The basic size includes the fields in the instance declared by the macro *PyObject_HEAD* or *PyObject_VAR_HEAD* (whichever is used to declare the instance struct) and this in turn includes the _ob_prev and _ob_next fields if they are present. This means that the only correct way to get an initializer for the *tp_basicsize* is to use the sizeof operator on the struct used to declare the instance layout. The basic size does not include the GC header size.

These fields are inherited separately by subtypes. If the base type has a non-zero *tp_itemsize*, it is generally not safe to set *tp_itemsize* to a different non-zero value in a subtype (though this depends on the implementation of the base type).

A note about alignment: if the variable items require a particular alignment, this should be taken care of by the value of *tp_basicsize*. Example: suppose a type implements an array of `double`. *tp_itemsize* is `sizeof(double)`. It is the programmer's responsibility that *tp_basicsize* is a multiple of `sizeof(double)` (assuming this is the alignment requirement for `double`).

destructor **PyTypeObject.tp_dealloc**
A pointer to the instance destructor function. This function must be defined unless the type guarantees that its instances will never be deallocated (as is the case for the singletons `None` and `Ellipsis`).

The destructor function is called by the *Py_DECREF()* and *Py_XDECREF()* macros when the new reference count is zero. At this point, the instance is still in existence, but there are no references to it. The destructor function should free all references which the instance owns, free all memory buffers owned by the instance (using the freeing function corresponding to the allocation function used to allocate the buffer), and finally (as its last action) call the type's *tp_free* function. If the type is not subtypable (doesn't have the *Py_TPFLAGS_BASETYPE* flag bit set), it is permissible to call the object deallocator directly instead of via *tp_free*. The object deallocator should be the one used to allocate the instance; this is normally *PyObject_Del()* if the instance was allocated using *PyObject_New()* or `PyObject_VarNew()`, or *PyObject_GC_Del()* if the instance was allocated using *PyObject_GC_New()* or *PyObject_GC_NewVar()*.

This field is inherited by subtypes.

printfunc **PyTypeObject.tp_print**
Reserved slot, formerly used for print formatting in Python 2.x.

getattrfunc **PyTypeObject.tp_getattr**
An optional pointer to the get-attribute-string function.

This field is deprecated. When it is defined, it should point to a function that acts the same as the *tp_getattro* function, but taking a C string instead of a Python string object to give the attribute name. The signature is

```
PyObject * tp_getattr(PyObject *o, char *attr_name);
```

This field is inherited by subtypes together with *tp_getattro*: a subtype inherits both *tp_getattr* and *tp_getattro* from its base type when the subtype's *tp_getattr* and *tp_getattro* are both `NULL`.

setattrfunc **PyTypeObject.tp_setattr**
An optional pointer to the function for setting and deleting attributes.

This field is deprecated. When it is defined, it should point to a function that acts the same as the *tp_setattro* function, but taking a C string instead of a Python string object to give the attribute name. The signature is

```
PyObject * tp_setattr(PyObject *o, char *attr_name, PyObject *v);
```

The *v* argument is set to `NULL` to delete the attribute. This field is inherited by subtypes together with *tp_setattro*: a subtype inherits both *tp_setattr* and *tp_setattro* from its base type when the subtype's *tp_setattr* and *tp_setattro* are both `NULL`.

*PyAsyncMethods** **tp_as_async**
Pointer to an additional structure that contains fields relevant only to objects which implement *awaitable* and *asynchronous iterator* protocols at the C-level. See *Async Object Structures* for details.

버전 3.5에 추가: Formerly known as `tp_compare` and `tp_reserved`.

reprfunc **PyTypeObject.tp_repr**
An optional pointer to a function that implements the built-in function `repr()`.

The signature is the same as for *PyObject_Repr()*; it must return a string or a Unicode object. Ideally, this function should return a string that, when passed to `eval()`, given a suitable environment, returns an object with the same value. If this is not feasible, it should return a string starting with `'<'` and ending with `'>'` from which both the type and the value of the object can be deduced.

When this field is not set, a string of the form `<%s object at %p>` is returned, where `%s` is replaced by the type name, and `%p` by the object's memory address.

This field is inherited by subtypes.

*PyNumberMethods** **tp_as_number**
Pointer to an additional structure that contains fields relevant only to objects which implement the number protocol. These fields are documented in *Number Object Structures*.

The `tp_as_number` field is not inherited, but the contained fields are inherited individually.

*PySequenceMethods** **tp_as_sequence**
Pointer to an additional structure that contains fields relevant only to objects which implement the sequence protocol. These fields are documented in *Sequence Object Structures*.

The `tp_as_sequence` field is not inherited, but the contained fields are inherited individually.

*PyMappingMethods** **tp_as_mapping**
Pointer to an additional structure that contains fields relevant only to objects which implement the mapping protocol. These fields are documented in *Mapping Object Structures*.

The `tp_as_mapping` field is not inherited, but the contained fields are inherited individually.

hashfunc **PyTypeObject.tp_hash**
An optional pointer to a function that implements the built-in function `hash()`.

The signature is the same as for *`PyObject_Hash()`*; it must return a value of the type Py_hash_t. The value `-1` should not be returned as a normal return value; when an error occurs during the computation of the hash value, the function should set an exception and return `-1`.

This field can be set explicitly to *`PyObject_HashNotImplemented()`* to block inheritance of the hash method from a parent type. This is interpreted as the equivalent of `__hash__ = None` at the Python level, causing `isinstance(o, collections.Hashable)` to correctly return `False`. Note that the converse is also true - setting `__hash__ = None` on a class at the Python level will result in the `tp_hash` slot being set to *`PyObject_HashNotImplemented()`*.

When this field is not set, an attempt to take the hash of the object raises `TypeError`.

This field is inherited by subtypes together with *`tp_richcompare`*: a subtype inherits both of *`tp_richcompare`* and *`tp_hash`*, when the subtype's *`tp_richcompare`* and *`tp_hash`* are both `NULL`.

ternaryfunc **PyTypeObject.tp_call**
An optional pointer to a function that implements calling the object. This should be `NULL` if the object is not callable. The signature is the same as for *`PyObject_Call()`*.

This field is inherited by subtypes.

reprfunc **PyTypeObject.tp_str**
An optional pointer to a function that implements the built-in operation `str()`. (Note that `str` is a type now, and `str()` calls the constructor for that type. This constructor calls *`PyObject_Str()`* to do the actual work, and *`PyObject_Str()`* will call this handler.)

The signature is the same as for *`PyObject_Str()`*; it must return a string or a Unicode object. This function should return a "friendly" string representation of the object, as this is the representation that will be used, among other things, by the `print()` function.

When this field is not set, *`PyObject_Repr()`* is called to return a string representation.

This field is inherited by subtypes.

getattrofunc **PyTypeObject.tp_getattro**
An optional pointer to the get-attribute function.

The signature is the same as for *`PyObject_GetAttr()`*. It is usually convenient to set this field to *`PyObject_GenericGetAttr()`*, which implements the normal way of looking for object attributes.

This field is inherited by subtypes together with *`tp_getattr`*: a subtype inherits both *`tp_getattr`* and *`tp_getattro`* from its base type when the subtype's *`tp_getattr`* and *`tp_getattro`* are both `NULL`.

setattrofunc **PyTypeObject.tp_setattro**
An optional pointer to the function for setting and deleting attributes.

The signature is the same as for *PyObject_SetAttr()*, but setting *v* to NULL to delete an attribute must be supported. It is usually convenient to set this field to *PyObject_GenericSetAttr()*, which implements the normal way of setting object attributes.

This field is inherited by subtypes together with *tp_setattr*: a subtype inherits both *tp_setattr* and *tp_setattro* from its base type when the subtype's *tp_setattr* and *tp_setattro* are both NULL.

*PyBufferProcs** **PyTypeObject.tp_as_buffer**
Pointer to an additional structure that contains fields relevant only to objects which implement the buffer interface. These fields are documented in *Buffer Object Structures*.

The *tp_as_buffer* field is not inherited, but the contained fields are inherited individually.

unsigned long **PyTypeObject.tp_flags**
This field is a bit mask of various flags. Some flags indicate variant semantics for certain situations; others are used to indicate that certain fields in the type object (or in the extension structures referenced via tp_as_number, tp_as_sequence, tp_as_mapping, and *tp_as_buffer*) that were historically not always present are valid; if such a flag bit is clear, the type fields it guards must not be accessed and must be considered to have a zero or NULL value instead.

Inheritance of this field is complicated. Most flag bits are inherited individually, i.e. if the base type has a flag bit set, the subtype inherits this flag bit. The flag bits that pertain to extension structures are strictly inherited if the extension structure is inherited, i.e. the base type's value of the flag bit is copied into the subtype together with a pointer to the extension structure. The *Py_TPFLAGS_HAVE_GC* flag bit is inherited together with the *tp_traverse* and *tp_clear* fields, i.e. if the *Py_TPFLAGS_HAVE_GC* flag bit is clear in the subtype and the *tp_traverse* and *tp_clear* fields in the subtype exist and have NULL values.

The following bit masks are currently defined; these can be ORed together using the | operator to form the value of the *tp_flags* field. The macro *PyType_HasFeature()* takes a type and a flags value, *tp* and *f*, and checks whether tp->tp_flags & f is non-zero.

Py_TPFLAGS_HEAPTYPE
This bit is set when the type object itself is allocated on the heap. In this case, the ob_type field of its instances is considered a reference to the type, and the type object is INCREF'ed when a new instance is created, and DECREF'ed when an instance is destroyed (this does not apply to instances of subtypes; only the type referenced by the instance's ob_type gets INCREF'ed or DECREF'ed).

Py_TPFLAGS_BASETYPE
This bit is set when the type can be used as the base type of another type. If this bit is clear, the type cannot be subtyped (similar to a "final" class in Java).

Py_TPFLAGS_READY
This bit is set when the type object has been fully initialized by *PyType_Ready()*.

Py_TPFLAGS_READYING
This bit is set while *PyType_Ready()* is in the process of initializing the type object.

Py_TPFLAGS_HAVE_GC
This bit is set when the object supports garbage collection. If this bit is set, instances must be created using *PyObject_GC_New()* and destroyed using *PyObject_GC_Del()*. More information in section 순환 가비지 수집 지원. This bit also implies that the GC-related fields *tp_traverse* and *tp_clear* are present in the type object.

Py_TPFLAGS_DEFAULT
This is a bitmask of all the bits that pertain to the existence of certain fields in the type object and its extension structures. Currently, it includes the following bits: Py_TPFLAGS_HAVE_STACKLESS_EXTENSION, Py_TPFLAGS_HAVE_VERSION_TAG.

Py_TPFLAGS_LONG_SUBCLASS

Py_TPFLAGS_LIST_SUBCLASS

Py_TPFLAGS_TUPLE_SUBCLASS

Py_TPFLAGS_BYTES_SUBCLASS

Py_TPFLAGS_UNICODE_SUBCLASS

Py_TPFLAGS_DICT_SUBCLASS

Py_TPFLAGS_BASE_EXC_SUBCLASS

Py_TPFLAGS_TYPE_SUBCLASS
These flags are used by functions such as *PyLong_Check()* to quickly determine if a type is a subclass of a built-in type; such specific checks are faster than a generic check, like *PyObject_IsInstance()*. Custom types that inherit from built-ins should have their *tp_flags* set appropriately, or the code that interacts with such types will behave differently depending on what kind of check is used.

Py_TPFLAGS_HAVE_FINALIZE
This bit is set when the *tp_finalize* slot is present in the type structure.

버전 3.4에 추가.

const char* **PyTypeObject.tp_doc**
An optional pointer to a NUL-terminated C string giving the docstring for this type object. This is exposed as the `__doc__` attribute on the type and instances of the type.

This field is *not* inherited by subtypes.

traverseproc **PyTypeObject.tp_traverse**
An optional pointer to a traversal function for the garbage collector. This is only used if the *Py_TPFLAGS_HAVE_GC* flag bit is set. More information about Python's garbage collection scheme can be found in section 순환 가비지 수집 지원.

The *tp_traverse* pointer is used by the garbage collector to detect reference cycles. A typical implementation of a *tp_traverse* function simply calls *Py_VISIT()* on each of the instance's members that are Python objects that the instance owns. For example, this is function `local_traverse()` from the `_thread` extension module:

```
static int
local_traverse(localobject *self, visitproc visit, void *arg)
{
    Py_VISIT(self->args);
    Py_VISIT(self->kw);
    Py_VISIT(self->dict);
    return 0;
}
```

Note that *Py_VISIT()* is called only on those members that can participate in reference cycles. Although there is also a `self->key` member, it can only be `NULL` or a Python string and therefore cannot be part of a reference cycle.

On the other hand, even if you know a member can never be part of a cycle, as a debugging aid you may want to visit it anyway just so the `gc` module's `get_referents()` function will include it.

경고: When implementing *tp_traverse*, only the members that the instance *owns* (by having strong references to them) must be visited. For instance, if an object supports weak references via the *tp_weaklist* slot, the pointer supporting the linked list (what *tp_weaklist* points to) must **not** be visited as the instance does not directly own the weak references to itself (the weakreference list is there to support the weak reference machinery, but the instance has no strong reference to the elements inside it, as they are allowed to be removed even if the instance is still alive).

Note that *Py_VISIT()* requires the *visit* and *arg* parameters to `local_traverse()` to have these specific names; don't name them just anything.

This field is inherited by subtypes together with *tp_clear* and the *Py_TPFLAGS_HAVE_GC* flag bit: the flag bit, *tp_traverse*, and *tp_clear* are all inherited from the base type if they are all zero in the subtype.

inquiry **PyTypeObject.tp_clear**
An optional pointer to a clear function for the garbage collector. This is only used if the *Py_TPFLAGS_HAVE_GC* flag bit is set.

The *tp_clear* member function is used to break reference cycles in cyclic garbage detected by the garbage collector. Taken together, all *tp_clear* functions in the system must combine to break all reference cycles. This is subtle, and if in any doubt supply a *tp_clear* function. For example, the tuple type does not implement a *tp_clear* function, because it's possible to prove that no reference cycle can be composed entirely of tuples. Therefore the *tp_clear* functions of other types must be sufficient to break any cycle containing a tuple. This isn't immediately obvious, and there's rarely a good reason to avoid implementing *tp_clear*.

Implementations of *tp_clear* should drop the instance's references to those of its members that may be Python objects, and set its pointers to those members to NULL, as in the following example:

```
static int
local_clear(localobject *self)
{
    Py_CLEAR(self->key);
    Py_CLEAR(self->args);
    Py_CLEAR(self->kw);
    Py_CLEAR(self->dict);
    return 0;
}
```

The *Py_CLEAR()* macro should be used, because clearing references is delicate: the reference to the contained object must not be decremented until after the pointer to the contained object is set to NULL. This is because decrementing the reference count may cause the contained object to become trash, triggering a chain of reclamation activity that may include invoking arbitrary Python code (due to finalizers, or weakref callbacks, associated with the contained object). If it's possible for such code to reference *self* again, it's important that the pointer to the contained object be NULL at that time, so that *self* knows the contained object can no longer be used. The *Py_CLEAR()* macro performs the operations in a safe order.

Because the goal of *tp_clear* functions is to break reference cycles, it's not necessary to clear contained objects like Python strings or Python integers, which can't participate in reference cycles. On the other hand, it may be convenient to clear all contained Python objects, and write the type's *tp_dealloc* function to invoke *tp_clear*.

More information about Python's garbage collection scheme can be found in section 순환 가비지 수집 지원.

This field is inherited by subtypes together with *tp_traverse* and the *Py_TPFLAGS_HAVE_GC* flag bit: the flag bit, *tp_traverse*, and *tp_clear* are all inherited from the base type if they are all zero in the subtype.

richcmpfunc **PyTypeObject.tp_richcompare**
An optional pointer to the rich comparison function, whose signature is PyObject *tp_richcompare(PyObject *a, PyObject *b, int op). The first parameter is guaranteed to be an instance of the type that is defined by *PyTypeObject*.

The function should return the result of the comparison (usually Py_True or Py_False). If the comparison is undefined, it must return Py_NotImplemented, if another error occurred it must return NULL and set an exception condition.

참고: If you want to implement a type for which only a limited set of comparisons makes sense (e.g. == and !=, but not < and friends), directly raise TypeError in the rich comparison function.

This field is inherited by subtypes together with *tp_hash*: a subtype inherits *tp_richcompare* and *tp_hash* when the subtype's *tp_richcompare* and *tp_hash* are both NULL.

The following constants are defined to be used as the third argument for *tp_richcompare* and for *PyObject_RichCompare()*:

Constant	Comparison
Py_LT	<
Py_LE	<=
Py_EQ	==
Py_NE	!=
Py_GT	>
Py_GE	>=

The following macro is defined to ease writing rich comparison functions:

PyObject ***Py_RETURN_RICHCOMPARE**(VAL_A, VAL_B, int *op*)

Return Py_True or Py_False from the function, depending on the result of a comparison. VAL_A and VAL_B must be orderable by C comparison operators (for example, they may be C ints or floats). The third argument specifies the requested operation, as for *PyObject_RichCompare()*.

The return value's reference count is properly incremented.

On error, sets an exception and returns NULL from the function.

버전 3.7에 추가.

Py_ssize_t **PyTypeObject.tp_weaklistoffset**

If the instances of this type are weakly referenceable, this field is greater than zero and contains the offset in the instance structure of the weak reference list head (ignoring the GC header, if present); this offset is used by PyObject_ClearWeakRefs() and the PyWeakref_*() functions. The instance structure needs to include a field of type *PyObject** which is initialized to NULL.

Do not confuse this field with *tp_weaklist*; that is the list head for weak references to the type object itself.

This field is inherited by subtypes, but see the rules listed below. A subtype may override this offset; this means that the subtype uses a different weak reference list head than the base type. Since the list head is always found via *tp_weaklistoffset*, this should not be a problem.

When a type defined by a class statement has no __slots__ declaration, and none of its base types are weakly referenceable, the type is made weakly referenceable by adding a weak reference list head slot to the instance layout and setting the *tp_weaklistoffset* of that slot's offset.

When a type's __slots__ declaration contains a slot named __weakref__, that slot becomes the weak reference list head for instances of the type, and the slot's offset is stored in the type's *tp_weaklistoffset*.

When a type's __slots__ declaration does not contain a slot named __weakref__, the type inherits its *tp_weaklistoffset* from its base type.

getiterfunc **PyTypeObject.tp_iter**

An optional pointer to a function that returns an iterator for the object. Its presence normally signals that the instances of this type are iterable (although sequences may be iterable without this function).

This function has the same signature as *PyObject_GetIter()*.

This field is inherited by subtypes.

iternextfunc **PyTypeObject.tp_iternext**

An optional pointer to a function that returns the next item in an iterator. When the iterator is exhausted, it must return NULL; a StopIteration exception may or may not be set. When another error occurs, it must return NULL too. Its presence signals that the instances of this type are iterators.

Iterator types should also define the *tp_iter* function, and that function should return the iterator instance itself (not a new iterator instance).

This function has the same signature as *PyIter_Next()*.

This field is inherited by subtypes.

struct *PyMethodDef** **PyTypeObject.tp_methods**
An optional pointer to a static NULL-terminated array of *PyMethodDef* structures, declaring regular methods of this type.

For each entry in the array, an entry is added to the type's dictionary (see *tp_dict* below) containing a method descriptor.

This field is not inherited by subtypes (methods are inherited through a different mechanism).

struct *PyMemberDef** **PyTypeObject.tp_members**
An optional pointer to a static NULL-terminated array of *PyMemberDef* structures, declaring regular data members (fields or slots) of instances of this type.

For each entry in the array, an entry is added to the type's dictionary (see *tp_dict* below) containing a member descriptor.

This field is not inherited by subtypes (members are inherited through a different mechanism).

struct *PyGetSetDef** **PyTypeObject.tp_getset**
An optional pointer to a static NULL-terminated array of *PyGetSetDef* structures, declaring computed attributes of instances of this type.

For each entry in the array, an entry is added to the type's dictionary (see *tp_dict* below) containing a getset descriptor.

This field is not inherited by subtypes (computed attributes are inherited through a different mechanism).

*PyTypeObject** **PyTypeObject.tp_base**
An optional pointer to a base type from which type properties are inherited. At this level, only single inheritance is supported; multiple inheritance require dynamically creating a type object by calling the metatype.

This field is not inherited by subtypes (obviously), but it defaults to `&PyBaseObject_Type` (which to Python programmers is known as the type `object`).

*PyObject** **PyTypeObject.tp_dict**
The type's dictionary is stored here by *PyType_Ready()*.

This field should normally be initialized to NULL before PyType_Ready is called; it may also be initialized to a dictionary containing initial attributes for the type. Once *PyType_Ready()* has initialized the type, extra attributes for the type may be added to this dictionary only if they don't correspond to overloaded operations (like `__add__()`).

This field is not inherited by subtypes (though the attributes defined in here are inherited through a different mechanism).

> **경고:** It is not safe to use *PyDict_SetItem()* on or otherwise modify *tp_dict* with the dictionary C-API.

descrgetfunc **PyTypeObject.tp_descr_get**
An optional pointer to a "descriptor get" function.

The function signature is

```
PyObject * tp_descr_get(PyObject *self, PyObject *obj, PyObject *type);
```

This field is inherited by subtypes.

descrsetfunc **PyTypeObject.tp_descr_set**
An optional pointer to a function for setting and deleting a descriptor's value.

The function signature is

```
int tp_descr_set(PyObject *self, PyObject *obj, PyObject *value);
```

The *value* argument is set to NULL to delete the value. This field is inherited by subtypes.

Py_ssize_t **PyTypeObject.tp_dictoffset**

If the instances of this type have a dictionary containing instance variables, this field is non-zero and contains the offset in the instances of the type of the instance variable dictionary; this offset is used by *PyObject_GenericGetAttr()*.

Do not confuse this field with *tp_dict*; that is the dictionary for attributes of the type object itself.

If the value of this field is greater than zero, it specifies the offset from the start of the instance structure. If the value is less than zero, it specifies the offset from the *end* of the instance structure. A negative offset is more expensive to use, and should only be used when the instance structure contains a variable-length part. This is used for example to add an instance variable dictionary to subtypes of `str` or `tuple`. Note that the *tp_basicsize* field should account for the dictionary added to the end in that case, even though the dictionary is not included in the basic object layout. On a system with a pointer size of 4 bytes, *tp_dictoffset* should be set to `-4` to indicate that the dictionary is at the very end of the structure.

The real dictionary offset in an instance can be computed from a negative *tp_dictoffset* as follows:

```
dictoffset = tp_basicsize + abs(ob_size)*tp_itemsize + tp_dictoffset
if dictoffset is not aligned on sizeof(void*):
    round up to sizeof(void*)
```

where *tp_basicsize*, *tp_itemsize* and *tp_dictoffset* are taken from the type object, and `ob_size` is taken from the instance. The absolute value is taken because ints use the sign of `ob_size` to store the sign of the number. (There's never a need to do this calculation yourself; it is done for you by `_PyObject_GetDictPtr()`.)

This field is inherited by subtypes, but see the rules listed below. A subtype may override this offset; this means that the subtype instances store the dictionary at a difference offset than the base type. Since the dictionary is always found via *tp_dictoffset*, this should not be a problem.

When a type defined by a class statement has no `__slots__` declaration, and none of its base types has an instance variable dictionary, a dictionary slot is added to the instance layout and the *tp_dictoffset* is set to that slot's offset.

When a type defined by a class statement has a `__slots__` declaration, the type inherits its *tp_dictoffset* from its base type.

(Adding a slot named `__dict__` to the `__slots__` declaration does not have the expected effect, it just causes confusion. Maybe this should be added as a feature just like `__weakref__` though.)

initproc **PyTypeObject.tp_init**

An optional pointer to an instance initialization function.

This function corresponds to the `__init__()` method of classes. Like `__init__()`, it is possible to create an instance without calling `__init__()`, and it is possible to reinitialize an instance by calling its `__init__()` method again.

The function signature is

```
int tp_init(PyObject *self, PyObject *args, PyObject *kwds)
```

The self argument is the instance to be initialized; the *args* and *kwds* arguments represent positional and keyword arguments of the call to `__init__()`.

The *tp_init* function, if not `NULL`, is called when an instance is created normally by calling its type, after the type's *tp_new* function has returned an instance of the type. If the *tp_new* function returns an instance of some other type that is not a subtype of the original type, no *tp_init* function is called; if *tp_new* returns an instance of a subtype of the original type, the subtype's *tp_init* is called.

This field is inherited by subtypes.

allocfunc **PyTypeObject.tp_alloc**

An optional pointer to an instance allocation function.

The function signature is

```
PyObject *tp_alloc(PyTypeObject *self, Py_ssize_t nitems)
```

The purpose of this function is to separate memory allocation from memory initialization. It should return a pointer to a block of memory of adequate length for the instance, suitably aligned, and initialized to zeros, but with `ob_refcnt` set to `1` and `ob_type` set to the type argument. If the type's *tp_itemsize* is non-zero, the object's `ob_size` field should be initialized to *nitems* and the length of the allocated memory block should be `tp_basicsize + nitems*tp_itemsize`, rounded up to a multiple of `sizeof(void*)`; otherwise, *nitems* is not used and the length of the block should be *tp_basicsize*.

Do not use this function to do any other instance initialization, not even to allocate additional memory; that should be done by *tp_new*.

This field is inherited by static subtypes, but not by dynamic subtypes (subtypes created by a class statement); in the latter, this field is always set to *PyType_GenericAlloc()*, to force a standard heap allocation strategy. That is also the recommended value for statically defined types.

newfunc **PyTypeObject.tp_new**
An optional pointer to an instance creation function.

If this function is `NULL` for a particular type, that type cannot be called to create new instances; presumably there is some other way to create instances, like a factory function.

The function signature is

```
PyObject *tp_new(PyTypeObject *subtype, PyObject *args, PyObject *kwds)
```

The subtype argument is the type of the object being created; the *args* and *kwds* arguments represent positional and keyword arguments of the call to the type. Note that subtype doesn't have to equal the type whose *tp_new* function is called; it may be a subtype of that type (but not an unrelated type).

The *tp_new* function should call `subtype->tp_alloc(subtype, nitems)` to allocate space for the object, and then do only as much further initialization as is absolutely necessary. Initialization that can safely be ignored or repeated should be placed in the *tp_init* handler. A good rule of thumb is that for immutable types, all initialization should take place in *tp_new*, while for mutable types, most initialization should be deferred to *tp_init*.

This field is inherited by subtypes, except it is not inherited by static types whose *tp_base* is `NULL` or `&PyBaseObject_Type`.

destructor **PyTypeObject.tp_free**
An optional pointer to an instance deallocation function. Its signature is `freefunc`:

```
void tp_free(void *)
```

An initializer that is compatible with this signature is *PyObject_Free()*.

This field is inherited by static subtypes, but not by dynamic subtypes (subtypes created by a class statement); in the latter, this field is set to a deallocator suitable to match *PyType_GenericAlloc()* and the value of the *Py_TPFLAGS_HAVE_GC* flag bit.

inquiry **PyTypeObject.tp_is_gc**
An optional pointer to a function called by the garbage collector.

The garbage collector needs to know whether a particular object is collectible or not. Normally, it is sufficient to look at the object's type's *tp_flags* field, and check the *Py_TPFLAGS_HAVE_GC* flag bit. But some types have a mixture of statically and dynamically allocated instances, and the statically allocated instances are not collectible. Such types should define this function; it should return `1` for a collectible instance, and `0` for a non-collectible instance. The signature is

```
int tp_is_gc(PyObject *self)
```

(The only example of this are types themselves. The metatype, *PyType_Type*, defines this function to distinguish between statically and dynamically allocated types.)

This field is inherited by subtypes.

*PyObject** **PyTypeObject.tp_bases**
Tuple of base types.

This is set for types created by a class statement. It should be `NULL` for statically defined types.

This field is not inherited.

*PyObject** **PyTypeObject.tp_mro**
Tuple containing the expanded set of base types, starting with the type itself and ending with `object`, in Method Resolution Order.

This field is not inherited; it is calculated fresh by *PyType_Ready()*.

destructor **PyTypeObject.tp_finalize**
An optional pointer to an instance finalization function. Its signature is `destructor`:

```
void tp_finalize(PyObject *)
```

If *tp_finalize* is set, the interpreter calls it once when finalizing an instance. It is called either from the garbage collector (if the instance is part of an isolated reference cycle) or just before the object is deallocated. Either way, it is guaranteed to be called before attempting to break reference cycles, ensuring that it finds the object in a sane state.

tp_finalize should not mutate the current exception status; therefore, a recommended way to write a non-trivial finalizer is:

```
static void
local_finalize(PyObject *self)
{
    PyObject *error_type, *error_value, *error_traceback;

    /* Save the current exception, if any. */
    PyErr_Fetch(&error_type, &error_value, &error_traceback);

    /* ... */

    /* Restore the saved exception. */
    PyErr_Restore(error_type, error_value, error_traceback);
}
```

For this field to be taken into account (even through inheritance), you must also set the *Py_TPFLAGS_HAVE_FINALIZE* flags bit.

This field is inherited by subtypes.

버전 3.4에 추가.

더 보기:

"Safe object finalization" (**PEP 442**)

*PyObject** **PyTypeObject.tp_cache**
Unused. Not inherited. Internal use only.

*PyObject** **PyTypeObject.tp_subclasses**
List of weak references to subclasses. Not inherited. Internal use only.

*PyObject** **PyTypeObject.tp_weaklist**
Weak reference list head, for weak references to this type object. Not inherited. Internal use only.

The remaining fields are only defined if the feature test macro `COUNT_ALLOCS` is defined, and are for internal use only. They are documented here for completeness. None of these fields are inherited by subtypes.

Py_ssize_t **PyTypeObject.tp_allocs**
Number of allocations.

Py_ssize_t **PyTypeObject.tp_frees**
Number of frees.

Py_ssize_t **PyTypeObject.tp_maxalloc**
Maximum simultaneously allocated objects.

*PyTypeObject** **PyTypeObject.tp_next**
Pointer to the next type object with a non-zero *tp_allocs* field.

Also, note that, in a garbage collected Python, tp_dealloc may be called from any Python thread, not just the thread which created the object (if the object becomes part of a refcount cycle, that cycle might be collected by a garbage collection on any thread). This is not a problem for Python API calls, since the thread on which tp_dealloc is called will own the Global Interpreter Lock (GIL). However, if the object being destroyed in turn destroys objects from some other C or C++ library, care should be taken to ensure that destroying those objects on the thread which called tp_dealloc will not violate any assumptions of the library.

11.4 Number Object Structures

PyNumberMethods
This structure holds pointers to the functions which an object uses to implement the number protocol. Each function is used by the function of similar name documented in the 숫자 프로토콜 section.

Here is the structure definition:

```
typedef struct {
     binaryfunc nb_add;
     binaryfunc nb_subtract;
     binaryfunc nb_multiply;
     binaryfunc nb_remainder;
     binaryfunc nb_divmod;
     ternaryfunc nb_power;
     unaryfunc nb_negative;
     unaryfunc nb_positive;
     unaryfunc nb_absolute;
     inquiry nb_bool;
     unaryfunc nb_invert;
     binaryfunc nb_lshift;
     binaryfunc nb_rshift;
     binaryfunc nb_and;
     binaryfunc nb_xor;
     binaryfunc nb_or;
     unaryfunc nb_int;
     void *nb_reserved;
     unaryfunc nb_float;

     binaryfunc nb_inplace_add;
     binaryfunc nb_inplace_subtract;
     binaryfunc nb_inplace_multiply;
     binaryfunc nb_inplace_remainder;
     ternaryfunc nb_inplace_power;
     binaryfunc nb_inplace_lshift;
     binaryfunc nb_inplace_rshift;
     binaryfunc nb_inplace_and;
     binaryfunc nb_inplace_xor;
     binaryfunc nb_inplace_or;

     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;
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    binaryfunc nb_inplace_matrix_multiply;
} PyNumberMethods;
```

참고: Binary and ternary functions must check the type of all their operands, and implement the necessary conversions (at least one of the operands is an instance of the defined type). If the operation is not defined for the given operands, binary and ternary functions must return `Py_NotImplemented`, if another error occurred they must return `NULL` and set an exception.

참고: The `nb_reserved` field should always be `NULL`. It was previously called `nb_long`, and was renamed in Python 3.0.1.

11.5 Mapping Object Structures

PyMappingMethods
This structure holds pointers to the functions which an object uses to implement the mapping protocol. It has three members:

lenfunc **PyMappingMethods.mp_length**
This function is used by *PyMapping_Size()* and *PyObject_Size()*, and has the same signature. This slot may be set to `NULL` if the object has no defined length.

binaryfunc **PyMappingMethods.mp_subscript**
This function is used by *PyObject_GetItem()* and *PySequence_GetSlice()*, and has the same signature as `PyObject_GetItem()`. This slot must be filled for the *PyMapping_Check()* function to return `1`, it can be `NULL` otherwise.

objobjargproc **PyMappingMethods.mp_ass_subscript**
This function is used by *PyObject_SetItem()*, *PyObject_DelItem()*, `PyObject_SetSlice()` and `PyObject_DelSlice()`. It has the same signature as `PyObject_SetItem()`, but *v* can also be set to `NULL` to delete an item. If this slot is `NULL`, the object does not support item assignment and deletion.

11.6 Sequence Object Structures

PySequenceMethods
This structure holds pointers to the functions which an object uses to implement the sequence protocol.

lenfunc **PySequenceMethods.sq_length**
This function is used by *PySequence_Size()* and *PyObject_Size()*, and has the same signature. It is also used for handling negative indices via the *sq_item* and the *sq_ass_item* slots.

binaryfunc **PySequenceMethods.sq_concat**
This function is used by *PySequence_Concat()* and has the same signature. It is also used by the `+` operator, after trying the numeric addition via the `nb_add` slot.

ssizeargfunc **PySequenceMethods.sq_repeat**
This function is used by *PySequence_Repeat()* and has the same signature. It is also used by the `*` operator, after trying numeric multiplication via the `nb_multiply` slot.

ssizeargfunc **PySequenceMethods.sq_item**
This function is used by *PySequence_GetItem()* and has the same signature. It is also used by *PyObject_GetItem()*, after trying the subscription via the *mp_subscript* slot. This slot must be filled for the *PySequence_Check()* function to return `1`, it can be `NULL` otherwise.

Negative indexes are handled as follows: if the `sq_length` slot is filled, it is called and the sequence length is used to compute a positive index which is passed to `sq_item`. If `sq_length` is `NULL`, the index is passed as is to the function.

ssizeobjargproc **`PySequenceMethods.sq_ass_item`**
This function is used by *`PySequence_SetItem()`* and has the same signature. It is also used by *`PyObject_SetItem()`* and *`PyObject_DelItem()`*, after trying the item assignment and deletion via the *`mp_ass_subscript`* slot. This slot may be left to `NULL` if the object does not support item assignment and deletion.

objobjproc **`PySequenceMethods.sq_contains`**
This function may be used by *`PySequence_Contains()`* and has the same signature. This slot may be left to `NULL`, in this case `PySequence_Contains()` simply traverses the sequence until it finds a match.

binaryfunc **`PySequenceMethods.sq_inplace_concat`**
This function is used by *`PySequence_InPlaceConcat()`* and has the same signature. It should modify its first operand, and return it. This slot may be left to `NULL`, in this case `PySequence_InPlaceConcat()` will fall back to *`PySequence_Concat()`*. It is also used by the augmented assignment `+=`, after trying numeric in-place addition via the `nb_inplace_add` slot.

ssizeargfunc **`PySequenceMethods.sq_inplace_repeat`**
This function is used by *`PySequence_InPlaceRepeat()`* and has the same signature. It should modify its first operand, and return it. This slot may be left to `NULL`, in this case `PySequence_InPlaceRepeat()` will fall back to *`PySequence_Repeat()`*. It is also used by the augmented assignment `*=`, after trying numeric in-place multiplication via the `nb_inplace_multiply` slot.

11.7 Buffer Object Structures

`PyBufferProcs`
This structure holds pointers to the functions required by the *Buffer protocol*. The protocol defines how an exporter object can expose its internal data to consumer objects.

getbufferproc **`PyBufferProcs.bf_getbuffer`**
The signature of this function is:

```
int (PyObject *exporter, Py_buffer *view, int flags);
```

Handle a request to *exporter* to fill in *view* as specified by *flags*. Except for point (3), an implementation of this function MUST take these steps:

(1) Check if the request can be met. If not, raise `PyExc_BufferError`, set `view->obj` to `NULL` and return `-1`.

(2) Fill in the requested fields.

(3) Increment an internal counter for the number of exports.

(4) Set `view->obj` to *exporter* and increment `view->obj`.

(5) Return `0`.

If *exporter* is part of a chain or tree of buffer providers, two main schemes can be used:

- Re-export: Each member of the tree acts as the exporting object and sets `view->obj` to a new reference to itself.
- Redirect: The buffer request is redirected to the root object of the tree. Here, `view->obj` will be a new reference to the root object.

The individual fields of *view* are described in section *Buffer structure*, the rules how an exporter must react to specific requests are in section *Buffer request types*.

All memory pointed to in the *Py_buffer* structure belongs to the exporter and must remain valid until there are no consumers left. *format*, *shape*, *strides*, *suboffsets* and *internal* are read-only for the consumer.

PyBuffer_FillInfo() provides an easy way of exposing a simple bytes buffer while dealing correctly with all request types.

PyObject_GetBuffer() is the interface for the consumer that wraps this function.

releasebufferproc **PyBufferProcs.bf_releasebuffer**
The signature of this function is:

```
void (PyObject *exporter, Py_buffer *view);
```

Handle a request to release the resources of the buffer. If no resources need to be released, *PyBufferProcs.bf_releasebuffer* may be `NULL`. Otherwise, a standard implementation of this function will take these optional steps:

(1) Decrement an internal counter for the number of exports.

(2) If the counter is `0`, free all memory associated with *view*.

The exporter MUST use the *internal* field to keep track of buffer-specific resources. This field is guaranteed to remain constant, while a consumer MAY pass a copy of the original buffer as the *view* argument.

This function MUST NOT decrement `view->obj`, since that is done automatically in *PyBuffer_Release()* (this scheme is useful for breaking reference cycles).

PyBuffer_Release() is the interface for the consumer that wraps this function.

11.8 Async Object Structures

버전 3.5에 추가.

PyAsyncMethods
This structure holds pointers to the functions required to implement *awaitable* and *asynchronous iterator* objects.

Here is the structure definition:

```
typedef struct {
    unaryfunc am_await;
    unaryfunc am_aiter;
    unaryfunc am_anext;
} PyAsyncMethods;
```

unaryfunc **PyAsyncMethods.am_await**
The signature of this function is:

```
PyObject *am_await(PyObject *self)
```

The returned object must be an iterator, i.e. *PyIter_Check()* must return `1` for it.

This slot may be set to `NULL` if an object is not an *awaitable*.

unaryfunc **PyAsyncMethods.am_aiter**
The signature of this function is:

```
PyObject *am_aiter(PyObject *self)
```

Must return an *awaitable* object. See `__anext__()` for details.

This slot may be set to `NULL` if an object does not implement asynchronous iteration protocol.

unaryfunc **PyAsyncMethods.am_anext**
The signature of this function is:

```
PyObject *am_anext(PyObject *self)
```

Must return an *awaitable* object. See `__anext__()` for details. This slot may be set to `NULL`.

11.9 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, 원자 형(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*)
Resize an object allocated by *PyObject_NewVar()*. Returns the resized object or `NULL` on failure. *op* must not be tracked by the collector yet.

void **PyObject_GC_Track**(*PyObject* *op)
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.

void **_PyObject_GC_TRACK**(*PyObject* *op)
*PyObject_GC_Track()*의 매크로 버전. 확장 모듈에는 사용하지 말아야 합니다.

버전 3.6부터 폐지: 이 매크로는 파이썬 3.8에서 삭제되었습니다.

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.

void **_PyObject_GC_UNTRACK**(*PyObject* *op)
*PyObject_GC_UnTrack()*의 매크로 버전. 확장 모듈에는 사용하지 말아야 합니다.

버전 3.6부터 폐지: 이 매크로는 파이썬 3.8에서 삭제되었습니다.

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)
Traversal function for a container object. Implementations must call the *visit* function for each object directly contained by *self*, with the parameters to *visit* being the contained object and the *arg* value passed to the handler. The *visit* function must not be called with a NULL object argument. If *visit* returns a non-zero value that value should be returned immediately.

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)
If *o* is not NULL, call the *visit* callback, with arguments *o* and *arg*. If *visit* returns a non-zero value, then return it. Using this macro, *tp_traverse* handlers look like:

```
static int
my_traverse(Noddy *self, visitproc visit, void *arg)
{
    Py_VISIT(self->foo);
    Py_VISIT(self->bar);
    return 0;
}
```

The *tp_clear* handler must be of the *inquiry* type, or NULL if the object is immutable.

int **(*inquiry)**(*PyObject* *self)
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 12

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 32 비트 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.h에 정의됩니다.

APPENDIX A

용어집

>>> 대화형 셸의 기본 파이썬 프롬프트. 인터프리터에서 대화형으로 실행될 수 있는 코드 예에서 자주 볼 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.

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-reference을 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 파인더와 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 매개변수나 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 `3` 과 `5` 는 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 차이 와 **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` 표현식 까지 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 이터레이터의 `__anext__()` 메서드가 돌려주는 어웨이터블을 풉니다. **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. 코루틴 이나 `__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 를 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.

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 `3`으로 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 필요하면, `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) Coroutines are a more generalized form of subroutines. Subroutines are entered at one point and exited at another point. Coroutines can be entered, exited, and resumed at many different points. They can be implemented with the `async def` statement. See also **PEP 492**.

coroutine function (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 정의 와 클래스 정의 의 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 딕셔너리에서 *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, 슈퍼클래스 참조 등의 많은 기능의 기초를 이루고 있기 때문입니다.

디스크립터의 메서드들에 대한 자세한 내용은 descriptors에 나옵니다.

dictionary (딕셔너리) 임의의 키를 값에 대응시키는 연관 배열 (associative array). 키는 `__hash__()` 와 `__eq__()` 메서드를 갖는 모든 객체가 될 수 있습니다. 펄에서 해시라고 부릅니다.

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-views를 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. 인트로스팩션을 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 꽥꽥댄다면, 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 클래스 로 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 문장 들이 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 리터럴 의 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 객체 의 비슷한 말.

finder (파인더) 임포트될 모듈을 위한 로더 를 찾으려고 시도하는 객체.

파이썬 3.3. 이후로, 두 종류의 파인더가 있습니다: `sys.meta_path` 와 함께 사용하는 메타 경로 파인더 와 `sys.path_hooks` 과 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 `2.75`를 돌려줍니다. `(-11) // 4` 가 `-2.75`를 내림 한 `-3`이 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 받아들일 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 이터레이터 를 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 이터레이터 가 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 코드 를 실행하도록 보장하기 위해 *CPython* 인터프리터가 사용하는 메커니즘. (`dict`와 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, 흔한 단일 프로세서 경우의 성능 저하가 심하기 때문입

니다. 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-invalidation을 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 파인더 가 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 (임포팅) 한 모듈의 파이썬 코드가 다른 모듈의 파이썬 코드에서 사용될 수 있도록 하는 절차.

importer (임포터) 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 구분이 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 여러분을 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 객

체는 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 클래스 는 `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 딕셔너리, 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.

metaclasses 에서 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 Order를 보면 됩니다.

module (모듈) 파이썬 코드의 조직화 단위를 담당하는 객체. 모듈은 임의의 파이썬 객체들을 담는 이름 공간을 갖습니다. 모듈은 임포팅 절차에 의해 파이썬으로 로드됩니다.

패키지 도 보세요.

module spec (모듈 스펙) 모듈을 로드하는데 사용되는 임포트 관련 정보들을 담고 있는 이름 공간. `importlib.machinery.ModuleSpec` 의 인스턴스.

MRO 메서드 결정 순서 를 보세요.

mutable (가변) 가변 객체는 값이 변할 수 있지만 `id()` 는 일정하게 유지합니다. 불변 도 보세요.

named tuple (네임드 튜플) The term "named tuple" applies to any type or class that inherits from tuple and whose indexable elements are also accessible using named attributes. The type or class may have other features as well.

Several built-in types are named tuples, including the values returned by `time.localtime()` and `os.stat()`. Another example is `sys.float_info`:

```
>>> sys.float_info[1]                    # indexed access
1024
>>> sys.float_info.max_exp               # named field access
1024
>>> isinstance(sys.float_info, tuple)    # kind of tuple
True
```

Some named tuples are built-in types (such as the above examples). Alternatively, a named tuple can be created from a regular class definition that inherits from `tuple` and that defines named fields. Such a class can be written by hand or it can be created with the factory function `collections.namedtuple()`. The latter technique also adds some extra methods that may not be found in hand-written or built-in named tuples.

namespace (이름 공간) 변수가 저장되는 장소. 이름 공간은 딕셔너리로 구현됩니다. 객체에 중첩된 이름 공간 (메서드 에서) 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.

모듈 도 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,

__getattribute__(), 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 클래스 의 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 인자 로 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. 파이썬은 위치-전용 매개변수를 정의하는 문법을 갖고 있지 않습니다. 하지만, 어떤 매장 함수들은 위치-전용 매개변수를 갖습니다 (예를 들어, abs()).
- 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 파인더 가 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 훅) 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 훅) 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 411을 보면 됩니다.

provisional package (잠정 패키지) 잠정 *API* 를 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 패키지 도 보세요.

__slots__ 클래스 내부의 선언인데, 인스턴스 어트리뷰트들을 위한 공간을 미리 선언하고 인스턴스 딕셔너리를 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.

collections.abc.Sequence 추상 베이스 클래스는 __getitem__() 과 __len__() 을 넘어서 훨씬 풍부한 인터페이스를 정의하는데, count(), index(), __contains__(), __reversed__() 를 추가합니다. 이 확장된 인터페이스를 구현한 형을 register() 를 사용해서 명시적으로 등록할 수 있습니다.

single dispatch (싱글 디스패치) 구현이 하나의 인자의 형에 기초해서 결정되는 제네릭 함수 디스패치의 한 형태.

slice (슬라이스) 보통 시퀀스 의 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 데이터스트림을 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 걸칠 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 힌트를 단순화하는 데 유용합니다. 예를 들면:

```
from typing import List, Tuple
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
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]

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, 인어를 이해하고 사용하는 데 도움이 됩니다. 이 목록은 대화형 프롬프트에서 "import this" 를 입력하면 보입니다.

APPENDIX B

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;
- reStructuredText와 Docutils 스위트를 만드는 Docutils 프로젝트.
- Fredrik Lundh, 그의 Alternative Python Reference 프로젝트에서 Sphinx가 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!

APPENDIX C

역사와 라이센스

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.

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs team moved to Digital Creations (now Zope Corporation; see https://www.zope.org/). In 2001, the Python Software Foundation (PSF, see https://www.python.org/psf/) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation is a sponsoring member of the PSF.

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 라이센스는 GPL과 달리 여러분의 변경을 공개 소스로 만들지 않고 수정된 버전을 배포할 수 있게

합니다. GPL 호환 라이센스는 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

C.2.1 PSF LICENSE AGREEMENT FOR PYTHON 3.7.17

```
1. This LICENSE AGREEMENT is between the Python Software Foundation␣
↪("PSF"), and
   the Individual or Organization ("Licensee") accessing and otherwise␣
↪using Python
   3.7.17 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 3.7.17 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 3.7.17 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 3.7.17 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
   3.7.17.

4. PSF is making Python 3.7.17 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 3.7.17 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.7.17
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A␣
↪RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.7.17, 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 3.7.17, 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.

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

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.3 포함된 소프트웨어에 대한 라이센스 및 승인

이 섹션은 파이썬 배포판에 포함된 제삼자 소프트웨어에 대한 불완전하지만 늘어나고 있는 라이센스와 승인의 목록입니다.

C.3.1 메르센 트위스터

_random 모듈은 http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html 에서 내려받은 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
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.


Any feedback is very welcome.
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)
```

C.3.2 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
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
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
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
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
  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
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
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
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.
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
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>

Original location:
   https://github.com/majek/csiphash/

Solution inspired by code from:
   Samuel Neves (supercop/crypto_auth/siphash24/little)
   djb (supercop/crypto_auth/siphash24/little2)
   Jean-Philippe Aumasson (https://131002.net/siphash/siphash24.c)
```

C.3.11 strtod 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 라이센스 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.
 *
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
* 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 라이센스 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.
   *
   * 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/)"
   *
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    * THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ``AS IS'' AND ANY
    * EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
    * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
    * PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE OpenSSL PROJECT OR
    * ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
    * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
    * NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
    * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
    * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
    * STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
    * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
    * OF THE POSSIBILITY OF SUCH DAMAGE.
    * ====================================================================
    *
    * This product includes cryptographic software written by Eric Young
    * (eay@cryptsoft.com).  This product includes software written by Tim
    * Hudson (tjh@cryptsoft.com).
    *
    */

Original SSLeay License
-----------------------

   /* Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com)
    * All rights reserved.
    *
    * This package is an SSL implementation written
    * by Eric Young (eay@cryptsoft.com).
    * The implementation was written so as to conform with Netscapes SSL.
    *
    * This library is free for commercial and non-commercial use as long as
    * the following conditions are aheared to.  The following conditions
    * apply to all code found in this distribution, be it the RC4, RSA,
    * 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)"
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
    *
    * 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,
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
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

`tracemalloc` 에 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:
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
  * Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

  * Redistributions in binary form must reproduce the above
    copyright notice, this list of conditions and the following
    disclaimer in the documentation and/or other materials provided
    with the distribution.

  * Neither the name of the author nor the names of its
    contributors may be used to endorse or promote products derived
    from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
OF THE POSSIBILITY OF SUCH DAMAGE.
```

C.3.17 libmpdec

_decimal 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.
```

APPENDIX D

저작권

파이썬과 이 설명서는:

전체 라이센스 및 사용 권한 정보는 역사와 라이센스 에서 제공합니다.

색인

Non-alphabetical

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

X

Y

Z