في ال **1.**

**oop**

**هنا نلاحظ وجود الدالة getName التي تقوم بإرجاع إسم الشخص, والدالة setName التي هي دالة غير نقية حيث تقوم بالتعديل على خاصية الإسم لدى الشخص**

**الدوال في ال functional programming**

**كلها نقية حيث تقوم بإستقبال المعطيات وإخراج ناتج دون وجود side effects أي أنها لا تقوم بالتعديل أو تغيير أحد القيم الموجودة لدي**

**تمتاز أنها declarative أي أن نقوم بمعالجة البيانات عبر مجموعة من الدوال الجاهزة بتجميع تلك الدوال معًا دون الإحتياج إلى كتابة الشفرة الخاصة بكل دالة منهم**

**كما تلاحظ في أننا هنا نقوم فقط بأخذ مجموعة من القيم الموجودة في مصفوفة ونقوم بالمرور عليها وطباعة العناصر دون أن نقوم بالتغيير في القيم الأصلية وكل سطر في البرنامج عبارة عن دالة**

**2.**

**Multithreading is the ability of a [program](https://www.techtarget.com/searchsoftwarequality/definition/program) or an [operating system](https://www.techtarget.com/whatis/definition/operating-system-OS) to enable more than one user at a time without requiring multiple copies of the program running on the computer. Multithreading can also handle multiple requests from the same user.**

**Each user request for a program or system service is tracked as a thread with a separate identity. As programs work on behalf of the initial thread request and are interrupted by other requests, the work status of the initial request is tracked until the work is completed. In this context, a user can also be another program.**

**Fast central processing unit ([CPU](https://www.techtarget.com/whatis/definition/processor)) speed and large [memory](https://www.techtarget.com/whatis/definition/memory) capacities are needed for multithreading. The single processor executes pieces, or threads, of various programs so fast, it appears the computer is handling multiple requests simultaneously.**

3.

**A multicore processor is an integrated circuit that has two or more processors attached for enhanced performance and reduced power consumption**

**4.**

**Tabulation is defined as the process of placing classified data in tabular form. A table is a systematic arrangement of statiscal information in rows and columns. The rows of a table are the horizontal arrangement of data whereas the columns of a table are the vertical arrangement of data.**

**5.**

****1. Use clear variable and function names.**** Code becomes much easier to read if you write out full, descriptive variable and function names.

****2. Write short functions that only do one thing.****Functions are more understandable, readable, and maintainable if they do one thing only. If there’s a bug when writing short functions, it is usually easier to find the source of that bug. Also, the code will be more reusable.

****3. Write good documentation**** for your code so that future developers understand what your code is doing and why.

****4. Be Consistent.****When writing code, consistency is key. People shouldn’t be able to look at a code base and tell exactly who wrote each line of code without a git blame! If you are using semicolons in JavaScript, use them at the end of each statement. Use ” vs ‘ consistently as well!

To begin with, you can set up a style guide and a linter to enforce these standards. Having those set up saves time when you’re writing code and makes sure you have some standard and consistency across your code base. For example, Ali recommends [Standard JS](https://standardjs.com/" \t "https://www.myhatchpad.com/insight/7-practical-tips-for-writing-clean-code/_blank) for JavaScript and [PEP8](https://www.python.org/dev/peps/pep-0008/?" \t "https://www.myhatchpad.com/insight/7-practical-tips-for-writing-clean-code/_blank) for Python. She’s also set up her text editor to enforce these standards when saving her work.

****5. Encapsulation + Modularization.****Group like variables and functions in order to make your code more reusable and understandable. Break long programs into different files so that your code is more modular and digestible. Long files are often hard to sift through, and you may want to use small chunks of code from project to project. Group like items in your code so that it is more reusable.

****6. Sandi Metz’s Rules.****Sandi Metz, Ruby developer, speaker, and author, has four rules for writing clean code in object oriented languages.

* Classes can be no longer than 100 lines of code
* Methods and functions can be no longer than 5 lines of code
* Pass no more than 4 parameters into a method
* Controllers can instantiate only one object

“At first it’s really challenging to follow these rules,” said Ali. “A lot of developers disagree with them and think they’re too restrictive.” But for Ali, adhering to these principles for several months allowed her to get in the right pattern of thinking. Now the concepts come naturally as she writes code.

To get more detail on Sandi Metz’s 4 principles, check out [her full talk](https://www.youtube.com/watch?v=npOGOmkxuio" \t "https://www.myhatchpad.com/insight/7-practical-tips-for-writing-clean-code/_blank)!

****7. The DRY Principle: Don’t Repeat Yourself.****This is one of the first things that developers are taught. “Early on, I think it’s a really great rule for why we’re writing code in the first place and how to think about code,” said Ali. “We’re writing for loops and functions so we don’t have to write the same piece of code over and over again.”

That being said, applying the DRY principle too much can lead to over abstraction. Ali explained that over abstraction can be especially common when developers get more advanced. “But early on, it’s helpful to think ‘Okay, am I doing similar code in 8 different places? If I am, I need to think about how to make this into a function.’”

6.

**A CRON script is a list of one or more commands to a computer operating system or application server that are to be executed at a specified time.**

**7.**

**There are different ways to fetch data in Python. For this post, we will use the requests module in Python. The requests module is a simple, yet elegant, HTTP library. To install this library**

**use the following command:**

**pip install requests**

**To check the installed version, use the following command:**

 pip freeze | grep requests  
 requests==2.22.0

**Looks like your environment is now ready to throw some requests.**

def get\_data(self, api):  
        response = requests.get(f"{api}")  
        if response.status\_code == 200:  
            print("sucessfully fetched the data")  
            self.formatted\_print(response.json())  
        else:  
            print(f"Hello person, there's a {response.status\_code} error with your request")

**As seen above, we first check the status code and then print the data. This code tells us about the response that has been received based on our requests. The 200 code tells us that we have received the info successfully.**

8.

## 1.1. A Simple Example

Let’s create an extension module called spam (the favorite food of Monty Python fans…) and let’s say we want to create a Python interface to the C library function system() [1](https://docs.python.org/3/extending/extending.html" \l "id5). This function takes a null-terminated character string as argument and returns an integer. We want this function to be callable from Python as follows:

>>>

**>>> import** **spam>>>** status = spam.system("ls -l")

Begin by creating a file spammodule.c. (Historically, if a module is called spam, the C file containing its implementation is called spammodule.c; if the module name is very long, like spammify, the module name can be just spammify.c.)

The first two lines of our file can be:

#define PY\_SSIZE\_T\_CLEAN#include *<Python.h>*

which pulls in the Python API (you can add a comment describing the purpose of the module and a copyright notice if you like).

**Note**

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 [Extracting Parameters in Extension Functions](https://docs.python.org/3/extending/extending.html" \l "parsetuple) for a description of this macro.

All user-visible symbols defined by Python.h have a prefix of Py or PY, except those defined in standard header files. For convenience, and since they are used extensively by the Python interpreter, "Python.h" includes a few standard header files: <stdio.h>, <string.h>, <errno.h>, and <stdlib.h>. If the latter header file does not exist on your system, it declares the functions malloc(), free() and realloc() directly.

The next thing we add to our module file is the C function that will be called when the Python expression spam.system(string) is evaluated (we’ll see shortly how it ends up being called):

**static** PyObject \*spam\_system(PyObject \*self, PyObject \*args){ **const** char \*command; int sts;

**if** (!PyArg\_ParseTuple(args, "s", &command)) **return** NULL; sts = system(command); **return** PyLong\_FromLong(sts);}

There is a straightforward translation from the argument list in Python (for example, the single expression "ls -l") to the arguments passed to the C function. The C function always has two arguments, conventionally named self and args.

The self argument points to the module object for module-level functions; for a method it would point to the object instance.

The args argument will be a pointer to a Python tuple object containing the arguments. Each item of the tuple corresponds to an argument in the call’s argument list. The arguments are Python objects — in order to do anything with them in our C function we have to convert them to C values. The function [PyArg\_ParseTuple()](https://docs.python.org/3/c-api/arg.html" \l "c.PyArg_ParseTuple" \o "PyArg_ParseTuple) in the Python API checks the argument types and converts them to C values. It uses a template string to determine the required types of the arguments as well as the types of the C variables into which to store the converted values. More about this later.

[PyArg\_ParseTuple()](https://docs.python.org/3/c-api/arg.html" \l "c.PyArg_ParseTuple" \o "PyArg_ParseTuple) returns true (nonzero) if all arguments have the right type and its components have been stored in the variables whose addresses are passed. It returns false (zero) if an invalid argument list was passed. In the latter case it also raises an appropriate exception so the calling function can return NULL immediately (as we saw in the example).

## 1.2. Intermezzo: Errors and Exceptions

An important convention throughout the Python interpreter is the following: when a function fails, it should set an exception condition and return an error value (usually -1 or a NULL pointer). Exception information is stored in three members of the interpreter’s thread state. These are NULL if there is no exception. Otherwise they are the C equivalents of the members of the Python tuple returned by [sys.exc\_info()](https://docs.python.org/3/library/sys.html" \l "sys.exc_info" \o "sys.exc_info). These are the exception type, exception instance, and a traceback object. It is important to know about them to understand how errors are passed around.

The Python API defines a number of functions to set various types of exceptions.

The most common one is [PyErr\_SetString()](https://docs.python.org/3/c-api/exceptions.html" \l "c.PyErr_SetString" \o "PyErr_SetString). Its arguments are an exception object and a C string. The exception object is usually a predefined object like PyExc\_ZeroDivisionError. The C string indicates the cause of the error and is converted to a Python string object and stored as the “associated value” of the exception.

Another useful function is [PyErr\_SetFromErrno()](https://docs.python.org/3/c-api/exceptions.html" \l "c.PyErr_SetFromErrno" \o "PyErr_SetFromErrno), which only takes an exception argument and constructs the associated value by inspection of the global variable errno. The most general function is [PyErr\_SetObject()](https://docs.python.org/3/c-api/exceptions.html" \l "c.PyErr_SetObject" \o "PyErr_SetObject), which takes two object arguments, the exception and its associated value. You don’t need to [Py\_INCREF()](https://docs.python.org/3/c-api/refcounting.html" \l "c.Py_INCREF" \o "Py_INCREF) the objects passed to any of these functions.

You can test non-destructively whether an exception has been set with [PyErr\_Occurred()](https://docs.python.org/3/c-api/exceptions.html" \l "c.PyErr_Occurred" \o "PyErr_Occurred). This returns the current exception object, or NULL if no exception has occurred. You normally don’t need to call [PyErr\_Occurred()](https://docs.python.org/3/c-api/exceptions.html" \l "c.PyErr_Occurred" \o "PyErr_Occurred) to see whether an error occurred in a function call, since you should be able to tell from the return value.

When a function f that calls another function g detects that the latter fails, f should itself return an error value (usually NULL or -1). It should not call one of the PyErr\_\* functions — one has already been called by g. f’s caller is then supposed to also return an error indication to its caller, again without calling PyErr\_\*, and so on — the most detailed cause of the error was already reported by the function that first detected it. Once the error reaches the Python interpreter’s main loop, this aborts the currently executing Python code and tries to find an exception handler specified by the Python programmer.

(There are situations where a module can actually give a more detailed error message by calling another PyErr\_\* function, and in such cases it is fine to do so. As a general rule, however, this is not necessary, and can cause information about the cause of the error to be lost: most operations can fail for a variety of reasons.)

To ignore an exception set by a function call that failed, the exception condition must be cleared explicitly by calling [PyErr\_Clear()](https://docs.python.org/3/c-api/exceptions.html" \l "c.PyErr_Clear" \o "PyErr_Clear). The only time C code should call [PyErr\_Clear()](https://docs.python.org/3/c-api/exceptions.html" \l "c.PyErr_Clear" \o "PyErr_Clear) is if it doesn’t want to pass the error on to the interpreter but wants to handle it completely by itself (possibly by trying something else, or pretending nothing went wrong).

Every failing malloc() call must be turned into an exception — the direct caller of malloc() (or realloc()) must call [PyErr\_NoMemory()](https://docs.python.org/3/c-api/exceptions.html" \l "c.PyErr_NoMemory" \o "PyErr_NoMemory) and return a failure indicator itself. All the object-creating functions (for example, [PyLong\_FromLong()](https://docs.python.org/3/c-api/long.html" \l "c.PyLong_FromLong" \o "PyLong_FromLong)) already do this, so this note is only relevant to those who call malloc() directly.

Also note that, with the important exception of [PyArg\_ParseTuple()](https://docs.python.org/3/c-api/arg.html" \l "c.PyArg_ParseTuple" \o "PyArg_ParseTuple) and friends, functions that return an integer status usually return a positive value or zero for success and -1 for failure, like Unix system calls.

Finally, be careful to clean up garbage (by making [Py\_XDECREF()](https://docs.python.org/3/c-api/refcounting.html" \l "c.Py_XDECREF" \o "Py_XDECREF) or [Py\_DECREF()](https://docs.python.org/3/c-api/refcounting.html" \l "c.Py_DECREF" \o "Py_DECREF) calls for objects you have already created) when you return an error indicator!

The choice of which exception to raise is entirely yours. There are predeclared C objects corresponding to all built-in Python exceptions, such as PyExc\_ZeroDivisionError, which you can use directly. Of course, you should choose exceptions wisely — don’t use PyExc\_TypeError to mean that a file couldn’t be opened (that should probably be PyExc\_IOError). If something’s wrong with the argument list, the [PyArg\_ParseTuple()](https://docs.python.org/3/c-api/arg.html" \l "c.PyArg_ParseTuple" \o "PyArg_ParseTuple) function usually raises PyExc\_TypeError. If you have an argument whose value must be in a particular range or must satisfy other conditions, PyExc\_ValueError is appropriate.

You can also define a new exception that is unique to your module. For this, you usually declare a static object variable at the beginning of your file:

**static** PyObject \*SpamError;

and initialize it in your module’s initialization function (PyInit\_spam()) with an exception object:

PyMODINIT\_FUNCPyInit\_spam(void){ PyObject \*m;

m = PyModule\_Create(&spammodule); **if** (m == NULL) **return** NULL;

SpamError = PyErr\_NewException("spam.error", NULL, NULL); Py\_XINCREF(SpamError); **if** (PyModule\_AddObject(m, "error", SpamError) < 0) { Py\_XDECREF(SpamError); Py\_CLEAR(SpamError); Py\_DECREF(m); **return** NULL; }

**return** m;}

Note that the Python name for the exception object is spam.error. The [PyErr\_NewException()](https://docs.python.org/3/c-api/exceptions.html" \l "c.PyErr_NewException" \o "PyErr_NewException) function may create a class with the base class being [Exception](https://docs.python.org/3/library/exceptions.html" \l "Exception" \o "Exception) (unless another class is passed in instead of NULL), described in [Built-in Exceptions](https://docs.python.org/3/library/exceptions.html" \l "bltin-exceptions).

Note also that the SpamError variable retains a reference to the newly created exception class; this is intentional! Since the exception could be removed from the module by external code, an owned reference to the class is needed to ensure that it will not be discarded, causing SpamError to become a dangling pointer. Should it become a dangling pointer, C code which raises the exception could cause a core dump or other unintended side effects.

We discuss the use of PyMODINIT\_FUNC as a function return type later in this sample.

The spam.error exception can be raised in your extension module using a call to [PyErr\_SetString()](https://docs.python.org/3/c-api/exceptions.html" \l "c.PyErr_SetString" \o "PyErr_SetString) as shown below:

**static** PyObject \*spam\_system(PyObject \*self, PyObject \*args){ **const** char \*command; int sts;

**if** (!PyArg\_ParseTuple(args, "s", &command)) **return** NULL; sts = system(command); **if** (sts < 0) { PyErr\_SetString(SpamError, "System command failed"); **return** NULL; } **return** PyLong\_FromLong(sts);}

## 1.3. Back to the Example

Going back to our example function, you should now be able to understand this statement:

**if** (!PyArg\_ParseTuple(args, "s", &command)) **return** NULL;

It returns NULL (the error indicator for functions returning object pointers) if an error is detected in the argument list, relying on the exception set by [PyArg\_ParseTuple()](https://docs.python.org/3/c-api/arg.html" \l "c.PyArg_ParseTuple" \o "PyArg_ParseTuple). Otherwise the string value of the argument has been copied to the local variable command. This is a pointer assignment and you are not supposed to modify the string to which it points (so in Standard C, the variable command should properly be declared as const char \*command).

The next statement is a call to the Unix function system(), passing it the string we just got from [PyArg\_ParseTuple()](https://docs.python.org/3/c-api/arg.html" \l "c.PyArg_ParseTuple" \o "PyArg_ParseTuple):

sts = system(command);

Our spam.system() function must return the value of sts as a Python object. This is done using the function [PyLong\_FromLong()](https://docs.python.org/3/c-api/long.html" \l "c.PyLong_FromLong" \o "PyLong_FromLong).

**return** PyLong\_FromLong(sts);

In this case, it will return an integer object. (Yes, even integers are objects on the heap in Python!)

If you have a C function that returns no useful argument (a function returning void), the corresponding Python function must return None. You need this idiom to do so (which is implemented by the [Py\_RETURN\_NONE](https://docs.python.org/3/c-api/none.html" \l "c.Py_RETURN_NONE" \o "Py_RETURN_NONE) macro):

Py\_INCREF(Py\_None);**return** Py\_None;

[Py\_None](https://docs.python.org/3/c-api/none.html" \l "c.Py_None" \o "Py_None) is the C name for the special Python object None. It is a genuine Python object rather than a NULL pointer, which means “error” in most contexts, as we have seen.

## 1.4. The Module’s Method Table and Initialization Function

I promised to show how spam\_system() is called from Python programs. First, we need to list its name and address in a “method table”:

**static** PyMethodDef SpamMethods[] = { ... {"system", spam\_system, METH\_VARARGS, "Execute a shell command."}, ... {NULL, NULL, 0, NULL} */\* Sentinel \*/*};

Note the third entry (METH\_VARARGS). This is a flag telling the interpreter the calling convention to be used for the C function. It should normally always be METH\_VARARGS or METH\_VARARGS | METH\_KEYWORDS; a value of 0 means that an obsolete variant of [PyArg\_ParseTuple()](https://docs.python.org/3/c-api/arg.html" \l "c.PyArg_ParseTuple" \o "PyArg_ParseTuple) is used.

When using only METH\_VARARGS, the function should expect the Python-level parameters to be passed in as a tuple acceptable for parsing via [PyArg\_ParseTuple()](https://docs.python.org/3/c-api/arg.html" \l "c.PyArg_ParseTuple" \o "PyArg_ParseTuple); more information on this function is provided below.

The METH\_KEYWORDS bit may be set in the third field if keyword arguments should be passed to the function. In this case, the C function should accept a third PyObject \* parameter which will be a dictionary of keywords. Use [PyArg\_ParseTupleAndKeywords()](https://docs.python.org/3/c-api/arg.html" \l "c.PyArg_ParseTupleAndKeywords" \o "PyArg_ParseTupleAndKeywords) to parse the arguments to such a function.

The method table must be referenced in the module definition structure:

**static** **struct** **PyModuleDef** spammodule = { PyModuleDef\_HEAD\_INIT, "spam", */\* name of module \*/* spam\_doc, */\* module documentation, may be NULL \*/* -1, */\* size of per-interpreter state of the module, or -1 if the module keeps state in global variables. \*/* SpamMethods};

This structure, in turn, must be passed to the interpreter in the module’s initialization function. The initialization function must be named PyInit\_name(), where name is the name of the module, and should be the only non-static item defined in the module file:

PyMODINIT\_FUNCPyInit\_spam(void){ **return** PyModule\_Create(&spammodule);}

Note that PyMODINIT\_FUNC declares the function as PyObject \* return type, declares any special linkage declarations required by the platform, and for C++ declares the function as extern "C".

When the Python program imports module spam for the first time, PyInit\_spam() is called. (See below for comments about embedding Python.) It calls [PyModule\_Create()](https://docs.python.org/3/c-api/module.html" \l "c.PyModule_Create" \o "PyModule_Create), which returns a module object, and inserts built-in function objects into the newly created module based upon the table (an array of [PyMethodDef](https://docs.python.org/3/c-api/structures.html" \l "c.PyMethodDef" \o "PyMethodDef) structures) found in the module definition. [PyModule\_Create()](https://docs.python.org/3/c-api/module.html" \l "c.PyModule_Create" \o "PyModule_Create) returns a pointer to the module object that it creates. It may abort with a fatal error for certain errors, or return NULL if the module could not be initialized satisfactorily. The init function must return the module object to its caller, so that it then gets inserted into sys.modules.

When embedding Python, the PyInit\_spam() function is not called automatically unless there’s an entry in the PyImport\_Inittab table. To add the module to the initialization table, use [PyImport\_AppendInittab()](https://docs.python.org/3/c-api/import.html" \l "c.PyImport_AppendInittab" \o "PyImport_AppendInittab), optionally followed by an import of the module:

intmain(int argc, char \*argv[]){ wchar\_t \*program = Py\_DecodeLocale(argv[0], NULL); **if** (program == NULL) { fprintf(stderr, "Fatal error: cannot decode argv[0]**\n**"); exit(1); }

*/\* Add a built-in module, before Py\_Initialize \*/* **if** (PyImport\_AppendInittab("spam", PyInit\_spam) == -1) { fprintf(stderr, "Error: could not extend in-built modules table**\n**"); exit(1); }

*/\* Pass argv[0] to the Python interpreter \*/* Py\_SetProgramName(program);

*/\* Initialize the Python interpreter. Required. If this step fails, it will be a fatal error. \*/* Py\_Initialize();

*/\* Optionally import the module; alternatively, import can be deferred until the embedded script imports it. \*/* PyObject \*pmodule = PyImport\_ImportModule("spam"); **if** (!pmodule) { PyErr\_Print(); fprintf(stderr, "Error: could not import module 'spam'**\n**"); }

...

PyMem\_RawFree(program); **return** 0;}

**Note**

Removing entries from sys.modules or importing compiled modules into multiple interpreters within a process (or following a fork() without an intervening exec()) can create problems for some extension modules. Extension module authors should exercise caution when initializing internal data structures.

A more substantial example module is included in the Python source distribution as Modules/xxmodule.c. This file may be used as a template or simply read as an example.

**Note**

Unlike our spam example, xxmodule uses multi-phase initialization (new in Python 3.5), where a PyModuleDef structure is returned from PyInit\_spam, and creation of the module is left to the import machinery. For details on multi-phase initialization, see [PEP 489](https://www.python.org/dev/peps/pep-0489).

## 1.5. Compilation and Linkage

There are two more things to do before you can use your new extension: compiling and linking it with the Python system. If you use dynamic loading, the details may depend on the style of dynamic loading your system uses; see the chapters about building extension modules (chapter [Building C and C++ Extensions](https://docs.python.org/3/extending/building.html" \l "building)) and additional information that pertains only to building on Windows (chapter [Building C and C++ Extensions on Windows](https://docs.python.org/3/extending/windows.html" \l "building-on-windows)) for more information about this.

If you can’t use dynamic loading, or if you want to make your module a permanent part of the Python interpreter, you will have to change the configuration setup and rebuild the interpreter. Luckily, this is very simple on Unix: just place your file (spammodule.c for example) in the Modules/ directory of an unpacked source distribution, add a line to the file Modules/Setup.local describing your file:

spam spammodule.o

and rebuild the interpreter by running **make** in the toplevel directory. You can also run **make** in the Modules/ subdirectory, but then you must first rebuild Makefile there by running ‘**make** Makefile’. (This is necessary each time you change the Setup file.)

If your module requires additional libraries to link with, these can be listed on the line in the configuration file as well, for instance:

spam spammodule.o -lX11

## 1.6. Calling Python Functions from C

So far we have concentrated on making C functions callable from Python. The reverse is also useful: calling Python functions from C. This is especially the case for libraries that support so-called “callback” functions. If a C interface makes use of callbacks, the equivalent Python often needs to provide a callback mechanism to the Python programmer; the implementation will require calling the Python callback functions from a C callback. Other uses are also imaginable.

Fortunately, the Python interpreter is easily called recursively, and there is a standard interface to call a Python function. (I won’t dwell on how to call the Python parser with a particular string as input — if you’re interested, have a look at the implementation of the [-c](https://docs.python.org/3/using/cmdline.html" \l "cmdoption-c) command line option in Modules/main.c from the Python source code.)

Calling a Python function is easy. First, the Python program must somehow pass you the Python function object. You should provide a function (or some other interface) to do this. When this function is called, save a pointer to the Python function object (be careful to [Py\_INCREF()](https://docs.python.org/3/c-api/refcounting.html" \l "c.Py_INCREF" \o "Py_INCREF) it!) in a global variable — or wherever you see fit. For example, the following function might be part of a module definition:

**static** PyObject \*my\_callback = NULL;

**static** PyObject \*my\_set\_callback(PyObject \*dummy, PyObject \*args){ PyObject \*result = NULL; PyObject \*temp;

**if** (PyArg\_ParseTuple(args, "O:set\_callback", &temp)) { **if** (!PyCallable\_Check(temp)) { PyErr\_SetString(PyExc\_TypeError, "parameter must be callable"); **return** NULL; } Py\_XINCREF(temp); */\* Add a reference to new callback \*/* Py\_XDECREF(my\_callback); */\* Dispose of previous callback \*/* my\_callback = temp; */\* Remember new callback \*/* */\* Boilerplate to return "None" \*/* Py\_INCREF(Py\_None); result = Py\_None; } **return** result;}

This function must be registered with the interpreter using the [METH\_VARARGS](https://docs.python.org/3/c-api/structures.html" \l "METH_VARARGS" \o "METH_VARARGS) flag; this is described in section [The Module’s Method Table and Initialization Function](https://docs.python.org/3/extending/extending.html" \l "methodtable). The [PyArg\_ParseTuple()](https://docs.python.org/3/c-api/arg.html" \l "c.PyArg_ParseTuple" \o "PyArg_ParseTuple) function and its arguments are documented in section [Extracting Parameters in Extension Functions](https://docs.python.org/3/extending/extending.html" \l "parsetuple).

The macros [Py\_XINCREF()](https://docs.python.org/3/c-api/refcounting.html" \l "c.Py_XINCREF" \o "Py_XINCREF) and [Py\_XDECREF()](https://docs.python.org/3/c-api/refcounting.html" \l "c.Py_XDECREF" \o "Py_XDECREF) increment/decrement the reference count of an object and are safe in the presence of NULL pointers (but note that temp will not be NULL in this context). More info on them in section [Reference Counts](https://docs.python.org/3/extending/extending.html" \l "refcounts).

Later, when it is time to call the function, you call the C function [PyObject\_CallObject()](https://docs.python.org/3/c-api/call.html" \l "c.PyObject_CallObject" \o "PyObject_CallObject). This function has two arguments, both pointers to arbitrary Python objects: the Python function, and the argument list. The argument list must always be a tuple object, whose length is the number of arguments. To call the Python function with no arguments, pass in NULL, or an empty tuple; to call it with one argument, pass a singleton tuple. [Py\_BuildValue()](https://docs.python.org/3/c-api/arg.html" \l "c.Py_BuildValue" \o "Py_BuildValue) returns a tuple when its format string consists of zero or more format codes between parentheses. For example:

int arg;PyObject \*arglist;PyObject \*result;...arg = 123;...*/\* Time to call the callback \*/*arglist = Py\_BuildValue("(i)", arg);result = PyObject\_CallObject(my\_callback, arglist);Py\_DECREF(arglist);

[PyObject\_CallObject()](https://docs.python.org/3/c-api/call.html" \l "c.PyObject_CallObject" \o "PyObject_CallObject) returns a Python object pointer: this is the return value of the Python function. [PyObject\_CallObject()](https://docs.python.org/3/c-api/call.html" \l "c.PyObject_CallObject" \o "PyObject_CallObject) is “reference-count-neutral” with respect to its arguments. In the example a new tuple was created to serve as the argument list, which is [Py\_DECREF()](https://docs.python.org/3/c-api/refcounting.html" \l "c.Py_DECREF" \o "Py_DECREF)-ed immediately after the [PyObject\_CallObject()](https://docs.python.org/3/c-api/call.html" \l "c.PyObject_CallObject" \o "PyObject_CallObject) call.

The return value of [PyObject\_CallObject()](https://docs.python.org/3/c-api/call.html" \l "c.PyObject_CallObject" \o "PyObject_CallObject) is “new”: either it is a brand new object, or it is an existing object whose reference count has been incremented. So, unless you want to save it in a global variable, you should somehow [Py\_DECREF()](https://docs.python.org/3/c-api/refcounting.html" \l "c.Py_DECREF" \o "Py_DECREF) the result, even (especially!) if you are not interested in its value.

Before you do this, however, it is important to check that the return value isn’t NULL. If it is, the Python function terminated by raising an exception. If the C code that called [PyObject\_CallObject()](https://docs.python.org/3/c-api/call.html" \l "c.PyObject_CallObject" \o "PyObject_CallObject) is called from Python, it should now return an error indication to its Python caller, so the interpreter can print a stack trace, or the calling Python code can handle the exception. If this is not possible or desirable, the exception should be cleared by calling [PyErr\_Clear()](https://docs.python.org/3/c-api/exceptions.html" \l "c.PyErr_Clear" \o "PyErr_Clear). For example:

**if** (result == NULL) **return** NULL; */\* Pass error back \*/*...use result...Py\_DECREF(result);

Depending on the desired interface to the Python callback function, you may also have to provide an argument list to [PyObject\_CallObject()](https://docs.python.org/3/c-api/call.html" \l "c.PyObject_CallObject" \o "PyObject_CallObject). In some cases the argument list is also provided by the Python program, through the same interface that specified the callback function. It can then be saved and used in the same manner as the function object. In other cases, you may have to construct a new tuple to pass as the argument list. The simplest way to do this is to call [Py\_BuildValue()](https://docs.python.org/3/c-api/arg.html" \l "c.Py_BuildValue" \o "Py_BuildValue). For example, if you want to pass an integral event code, you might use the following code:

PyObject \*arglist;...arglist = Py\_BuildValue("(l)", eventcode);result = PyObject\_CallObject(my\_callback, arglist);Py\_DECREF(arglist);**if** (result == NULL) **return** NULL; */\* Pass error back \*//\* Here maybe use the result \*/*Py\_DECREF(result);

Note the placement of Py\_DECREF(arglist) immediately after the call, before the error check! Also note that strictly speaking this code is not complete: [Py\_BuildValue()](https://docs.python.org/3/c-api/arg.html" \l "c.Py_BuildValue" \o "Py_BuildValue) may run out of memory, and this should be checked.

You may also call a function with keyword arguments by using [PyObject\_Call()](https://docs.python.org/3/c-api/call.html" \l "c.PyObject_Call" \o "PyObject_Call), which supports arguments and keyword arguments. As in the above example, we use [Py\_BuildValue()](https://docs.python.org/3/c-api/arg.html" \l "c.Py_BuildValue" \o "Py_BuildValue) to construct the dictionary.

PyObject \*dict;...dict = Py\_BuildValue("{s:i}", "name", val);result = PyObject\_Call(my\_callback, NULL, dict);Py\_DECREF(dict);**if** (result == NULL) **return** NULL; */\* Pass error back \*//\* Here maybe use the result \*/*Py\_DECREF(result);

## 1.7. Extracting Parameters in Extension Functions

The [PyArg\_ParseTuple()](https://docs.python.org/3/c-api/arg.html" \l "c.PyArg_ParseTuple" \o "PyArg_ParseTuple) function is declared as follows:

int PyArg\_ParseTuple(PyObject \*arg, **const** char \*format, ...);

The arg argument must be a tuple object containing an argument list passed from Python to a C function. The format argument must be a format string, whose syntax is explained in [Parsing arguments and building values](https://docs.python.org/3/c-api/arg.html" \l "arg-parsing) in the Python/C API Reference Manual. The remaining arguments must be addresses of variables whose type is determined by the format string.

Note that while [PyArg\_ParseTuple()](https://docs.python.org/3/c-api/arg.html" \l "c.PyArg_ParseTuple" \o "PyArg_ParseTuple) checks that the Python arguments have the required types, it cannot check the validity of the addresses of C variables passed to the call: if you make mistakes there, your code will probably crash or at least overwrite random bits in memory. So be careful!

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

Some example calls:

#define PY\_SSIZE\_T\_CLEAN */\* Make "s#" use Py\_ssize\_t rather than int. \*/*#include *<Python.h>*

int ok;int i, j;long k, l;**const** char \*s;Py\_ssize\_t size;

ok = PyArg\_ParseTuple(args, ""); */\* No arguments \*/* */\* Python call: f() \*/*

ok = PyArg\_ParseTuple(args, "s", &s); */\* A string \*/* */\* Possible Python call: f('whoops!') \*/*

ok = PyArg\_ParseTuple(args, "lls", &k, &l, &s); */\* Two longs and a string \*/* */\* Possible Python call: f(1, 2, 'three') \*/*

ok = PyArg\_ParseTuple(args, "(ii)s#", &i, &j, &s, &size); */\* A pair of ints and a string, whose size is also returned \*/* */\* Possible Python call: f((1, 2), 'three') \*/*

{ **const** char \*file; **const** char \*mode = "r"; int bufsize = 0; ok = PyArg\_ParseTuple(args, "s|si", &file, &mode, &bufsize); */\* A string, and optionally another string and an integer \*/* */\* Possible Python calls: f('spam') f('spam', 'w') f('spam', 'wb', 100000) \*/*}

{ int left, top, right, bottom, h, v; ok = PyArg\_ParseTuple(args, "((ii)(ii))(ii)", &left, &top, &right, &bottom, &h, &v); */\* A rectangle and a point \*/* */\* Possible Python call: f(((0, 0), (400, 300)), (10, 10)) \*/*}

{ Py\_complex c; ok = PyArg\_ParseTuple(args, "D:myfunction", &c); */\* a complex, also providing a function name for errors \*/* */\* Possible Python call: myfunction(1+2j) \*/*}

## 1.8. Keyword Parameters for Extension Functions

The [PyArg\_ParseTupleAndKeywords()](https://docs.python.org/3/c-api/arg.html" \l "c.PyArg_ParseTupleAndKeywords" \o "PyArg_ParseTupleAndKeywords) function is declared as follows:

int PyArg\_ParseTupleAndKeywords(PyObject \*arg, PyObject \*kwdict, **const** char \*format, char \*kwlist[], ...);

The arg and format parameters are identical to those of the [PyArg\_ParseTuple()](https://docs.python.org/3/c-api/arg.html" \l "c.PyArg_ParseTuple" \o "PyArg_ParseTuple) function. The kwdict parameter is the dictionary of keywords received as the third parameter from the Python runtime. The kwlist parameter is a NULL-terminated list of strings which identify the parameters; the names are matched with the type information from format from left to right. On success, [PyArg\_ParseTupleAndKeywords()](https://docs.python.org/3/c-api/arg.html" \l "c.PyArg_ParseTupleAndKeywords" \o "PyArg_ParseTupleAndKeywords) returns true, otherwise it returns false and raises an appropriate exception.

**Note**

Nested tuples cannot be parsed when using keyword arguments! Keyword parameters passed in which are not present in the kwlist will cause [TypeError](https://docs.python.org/3/library/exceptions.html" \l "TypeError" \o "TypeError) to be raised.

Here is an example module which uses keywords, based on an example by Geoff Philbrick ([philbrick@hks.com](mailto:philbrick@hks.com)):

#define PY\_SSIZE\_T\_CLEAN */\* Make "s#" use Py\_ssize\_t rather than int. \*/*#include *<Python.h>*

**static** PyObject \*keywdarg\_parrot(PyObject \*self, PyObject \*args, PyObject \*keywds){ int voltage; **const** char \*state = "a stiff"; **const** char \*action = "voom"; **const** char \*type = "Norwegian Blue";

**static** char \*kwlist[] = {"voltage", "state", "action", "type", NULL};

**if** (!PyArg\_ParseTupleAndKeywords(args, keywds, "i|sss", kwlist, &voltage, &state, &action, &type)) **return** NULL;

printf("-- This parrot wouldn't %s if you put %i Volts through it.**\n**", action, voltage); printf("-- Lovely plumage, the %s -- It's %s!**\n**", type, state);

Py\_RETURN\_NONE;}

**static** PyMethodDef keywdarg\_methods[] = { */\* The cast of the function is necessary since PyCFunction values \* only take two PyObject\* parameters, and keywdarg\_parrot() takes \* three. \*/* {"parrot", (PyCFunction)(void(\*)(void))keywdarg\_parrot, METH\_VARARGS | METH\_KEYWORDS, "Print a lovely skit to standard output."}, {NULL, NULL, 0, NULL} */\* sentinel \*/*};

**static** **struct** **PyModuleDef** keywdargmodule = { PyModuleDef\_HEAD\_INIT, "keywdarg", NULL, -1, keywdarg\_methods};

PyMODINIT\_FUNCPyInit\_keywdarg(void){ **return** PyModule\_Create(&keywdargmodule);}

## 1.9. Building Arbitrary Values

This function is the counterpart to [PyArg\_ParseTuple()](https://docs.python.org/3/c-api/arg.html" \l "c.PyArg_ParseTuple" \o "PyArg_ParseTuple). It is declared as follows:

PyObject \*Py\_BuildValue(**const** char \*format, ...);

It recognizes a set of format units similar to the ones recognized by [PyArg\_ParseTuple()](https://docs.python.org/3/c-api/arg.html" \l "c.PyArg_ParseTuple" \o "PyArg_ParseTuple), but the arguments (which are input to the function, not output) must not be pointers, just values. It returns a new Python object, suitable for returning from a C function called from Python.

One difference with [PyArg\_ParseTuple()](https://docs.python.org/3/c-api/arg.html" \l "c.PyArg_ParseTuple" \o "PyArg_ParseTuple): while the latter requires its first argument to be a tuple (since Python argument lists are always represented as tuples internally), [Py\_BuildValue()](https://docs.python.org/3/c-api/arg.html" \l "c.Py_BuildValue" \o "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.

Examples (to the left the call, to the right the resulting Python value):

Py\_BuildValue("") None

Py\_BuildValue("i", 123) 123

Py\_BuildValue("iii", 123, 456, 789) (123, 456, 789)

Py\_BuildValue("s", "hello") 'hello'

Py\_BuildValue("y", "hello") b'hello'

Py\_BuildValue("ss", "hello", "world") ('hello', 'world')

Py\_BuildValue("s#", "hello", 4) 'hell'

Py\_BuildValue("y#", "hello", 4) b'hell'

Py\_BuildValue("()") ()

Py\_BuildValue("(i)", 123) (123,)

Py\_BuildValue("(ii)", 123, 456) (123, 456)

Py\_BuildValue("(i,i)", 123, 456) (123, 456)

Py\_BuildValue("[i,i]", 123, 456) [123, 456]

Py\_BuildValue("{s:i,s:i}",

"abc", 123, "def", 456) {'abc': 123, 'def': 456}

Py\_BuildValue("((ii)(ii)) (ii)",

1, 2, 3, 4, 5, 6) (((1, 2), (3, 4)), (5, 6))

## 1.10. Reference Counts

In languages like C or C++, the programmer is responsible for dynamic allocation and deallocation of memory on the heap. In C, this is done using the functions malloc() and free(). In C++, the operators new and delete are used with essentially the same meaning and we’ll restrict the following discussion to the C case.

Every block of memory allocated with malloc() should eventually be returned to the pool of available memory by exactly one call to free(). It is important to call free() at the right time. If a block’s address is forgotten but free() is not called for it, the memory it occupies cannot be reused until the program terminates. This is called a memory leak. On the other hand, if a program calls free() for a block and then continues to use the block, it creates a conflict with re-use of the block through another malloc() call. This is called using freed memory. It has the same bad consequences as referencing uninitialized data — core dumps, wrong results, mysterious crashes.

Common causes of memory leaks are unusual paths through the code. For instance, a function may allocate a block of memory, do some calculation, and then free the block again. Now a change in the requirements for the function may add a test to the calculation that detects an error condition and can return prematurely from the function. It’s easy to forget to free the allocated memory block when taking this premature exit, especially when it is added later to the code. Such leaks, once introduced, often go undetected for a long time: the error exit is taken only in a small fraction of all calls, and most modern machines have plenty of virtual memory, so the leak only becomes apparent in a long-running process that uses the leaking function frequently. Therefore, it’s important to prevent leaks from happening by having a coding convention or strategy that minimizes this kind of errors.

Since Python makes heavy use of malloc() and free(), it needs a strategy to avoid memory leaks as well as the use of freed memory. The chosen method is called reference counting. The principle is simple: every object contains a counter, which is incremented when a reference to the object is stored somewhere, and which is decremented when a reference to it is deleted. When the counter reaches zero, the last reference to the object has been deleted and the object is freed.

An alternative strategy is called automatic garbage collection. (Sometimes, reference counting is also referred to as a garbage collection strategy, hence my use of “automatic” to distinguish the two.) The big advantage of automatic garbage collection is that the user doesn’t need to call free() explicitly. (Another claimed advantage is an improvement in speed or memory usage — this is no hard fact however.) The disadvantage is that for C, there is no truly portable automatic garbage collector, while reference counting can be implemented portably (as long as the functions malloc() and free() are available — which the C Standard guarantees). Maybe some day a sufficiently portable automatic garbage collector will be available for C. Until then, we’ll have to live with reference counts.

While Python uses the traditional reference counting implementation, it also offers a cycle detector that works to detect reference cycles. This allows applications to not worry about creating direct or indirect circular references; these are the weakness of garbage collection implemented using only reference counting. Reference cycles consist of objects which contain (possibly indirect) references to themselves, so that each object in the cycle has a reference count which is non-zero. Typical reference counting implementations are not able to reclaim the memory belonging to any objects in a reference cycle, or referenced from the objects in the cycle, even though there are no further references to the cycle itself.

The cycle detector is able to detect garbage cycles and can reclaim them. The [gc](https://docs.python.org/3/library/gc.html" \l "module-gc" \o "gc: Interface to the cycle-detecting garbage collector.) module exposes a way to run the detector (the [collect()](https://docs.python.org/3/library/gc.html" \l "gc.collect" \o "gc.collect) function), as well as configuration interfaces and the ability to disable the detector at runtime.

### 1.10.1. Reference Counting in Python

There are two macros, Py\_INCREF(x) and Py\_DECREF(x), which handle the incrementing and decrementing of the reference count. [Py\_DECREF()](https://docs.python.org/3/c-api/refcounting.html" \l "c.Py_DECREF" \o "Py_DECREF) also frees the object when the count reaches zero. For flexibility, it doesn’t call free() directly — rather, it makes a call through a function pointer in the object’s type object. For this purpose (and others), every object also contains a pointer to its type object.

The big question now remains: when to use Py\_INCREF(x) and Py\_DECREF(x)? Let’s first introduce some terms. Nobody “owns” an object; however, you can own a reference to an object. An object’s reference count is now defined as the number of owned references to it. The owner of a reference is responsible for calling [Py\_DECREF()](https://docs.python.org/3/c-api/refcounting.html" \l "c.Py_DECREF" \o "Py_DECREF) when the reference is no longer needed. Ownership of a reference can be transferred. There are three ways to dispose of an owned reference: pass it on, store it, or call [Py\_DECREF()](https://docs.python.org/3/c-api/refcounting.html" \l "c.Py_DECREF" \o "Py_DECREF). Forgetting to dispose of an owned reference creates a memory leak.

It is also possible to borrow [2](https://docs.python.org/3/extending/extending.html" \l "id6) a reference to an object. The borrower of a reference should not call [Py\_DECREF()](https://docs.python.org/3/c-api/refcounting.html" \l "c.Py_DECREF" \o "Py_DECREF). The borrower must not hold on to the object longer than the owner from which it was borrowed. Using a borrowed reference after the owner has disposed of it risks using freed memory and should be avoided completely [3](https://docs.python.org/3/extending/extending.html" \l "id7).

The advantage of borrowing over owning a reference is that you don’t need to take care of disposing of the reference on all possible paths through the code — in other words, with a borrowed reference you don’t run the risk of leaking when a premature exit is taken. The disadvantage of borrowing over owning is that there are some subtle situations where in seemingly correct code a borrowed reference can be used after the owner from which it was borrowed has in fact disposed of it.

A borrowed reference can be changed into an owned reference by calling [Py\_INCREF()](https://docs.python.org/3/c-api/refcounting.html" \l "c.Py_INCREF" \o "Py_INCREF). This does not affect the status of the owner from which the reference was borrowed — it creates a new owned reference, and gives full owner responsibilities (the new owner must dispose of the reference properly, as well as the previous owner).

### 1.10.2. Ownership Rules

Whenever an object reference is passed into or out of a function, it is part of the function’s interface specification whether ownership is transferred with the reference or not.

Most functions that return a reference to an object pass on ownership with the reference. In particular, all functions whose function it is to create a new object, such as [PyLong\_FromLong()](https://docs.python.org/3/c-api/long.html" \l "c.PyLong_FromLong" \o "PyLong_FromLong) and [Py\_BuildValue()](https://docs.python.org/3/c-api/arg.html" \l "c.Py_BuildValue" \o "Py_BuildValue), pass ownership to the receiver. Even if the object is not actually new, you still receive ownership of a new reference to that object. For instance, [PyLong\_FromLong()](https://docs.python.org/3/c-api/long.html" \l "c.PyLong_FromLong" \o "PyLong_FromLong) maintains a cache of popular values and can return a reference to a cached item.

Many functions that extract objects from other objects also transfer ownership with the reference, for instance [PyObject\_GetAttrString()](https://docs.python.org/3/c-api/object.html" \l "c.PyObject_GetAttrString" \o "PyObject_GetAttrString). The picture is less clear, here, however, since a few common routines are exceptions: [PyTuple\_GetItem()](https://docs.python.org/3/c-api/tuple.html" \l "c.PyTuple_GetItem" \o "PyTuple_GetItem), [PyList\_GetItem()](https://docs.python.org/3/c-api/list.html" \l "c.PyList_GetItem" \o "PyList_GetItem), [PyDict\_GetItem()](https://docs.python.org/3/c-api/dict.html" \l "c.PyDict_GetItem" \o "PyDict_GetItem), and [PyDict\_GetItemString()](https://docs.python.org/3/c-api/dict.html" \l "c.PyDict_GetItemString" \o "PyDict_GetItemString) all return references that you borrow from the tuple, list or dictionary.

The function [PyImport\_AddModule()](https://docs.python.org/3/c-api/import.html" \l "c.PyImport_AddModule" \o "PyImport_AddModule) also returns a borrowed reference, even though it may actually create the object it returns: this is possible because an owned reference to the object is stored in sys.modules.

When you pass an object reference into another function, in general, the function borrows the reference from you — if it needs to store it, it will use [Py\_INCREF()](https://docs.python.org/3/c-api/refcounting.html" \l "c.Py_INCREF" \o "Py_INCREF) to become an independent owner. There are exactly two important exceptions to this rule: [PyTuple\_SetItem()](https://docs.python.org/3/c-api/tuple.html" \l "c.PyTuple_SetItem" \o "PyTuple_SetItem) and [PyList\_SetItem()](https://docs.python.org/3/c-api/list.html" \l "c.PyList_SetItem" \o "PyList_SetItem). These functions take over ownership of the item passed to them — even if they fail! (Note that [PyDict\_SetItem()](https://docs.python.org/3/c-api/dict.html" \l "c.PyDict_SetItem" \o "PyDict_SetItem) and friends don’t take over ownership — they are “normal.”)

When a C function is called from Python, it borrows references to its arguments from the caller. The caller owns a reference to the object, so the borrowed reference’s lifetime is guaranteed until the function returns. Only when such a borrowed reference must be stored or passed on, it must be turned into an owned reference by calling [Py\_INCREF()](https://docs.python.org/3/c-api/refcounting.html" \l "c.Py_INCREF" \o "Py_INCREF).

The object reference returned from a C function that is called from Python must be an owned reference — ownership is transferred from the function to its caller.

### 1.10.3. Thin Ice

There are a few situations where seemingly harmless use of a borrowed reference can lead to problems. These all have to do with implicit invocations of the interpreter, which can cause the owner of a reference to dispose of it.

The first and most important case to know about is using [Py\_DECREF()](https://docs.python.org/3/c-api/refcounting.html" \l "c.Py_DECREF" \o "Py_DECREF) on an unrelated object while borrowing a reference to a list item. For instance:

voidbug(PyObject \*list){ PyObject \*item = PyList\_GetItem(list, 0);

PyList\_SetItem(list, 1, PyLong\_FromLong(0L)); PyObject\_Print(item, stdout, 0); */\* BUG! \*/*}

This function first borrows a reference to list[0], then replaces list[1] with the value 0, and finally prints the borrowed reference. Looks harmless, right? But it’s not!

Let’s follow the control flow into [PyList\_SetItem()](https://docs.python.org/3/c-api/list.html" \l "c.PyList_SetItem" \o "PyList_SetItem). The list owns references to all its items, so when item 1 is replaced, it has to dispose of the original item 1. Now let’s suppose the original item 1 was an instance of a user-defined class, and let’s further suppose that the class defined a \_\_del\_\_() method. If this class instance has a reference count of 1, disposing of it will call its \_\_del\_\_() method.

Since it is written in Python, the \_\_del\_\_() method can execute arbitrary Python code. Could it perhaps do something to invalidate the reference to item in bug()? You bet! Assuming that the list passed into bug() is accessible to the \_\_del\_\_() method, it could execute a statement to the effect of del list[0], and assuming this was the last reference to that object, it would free the memory associated with it, thereby invalidating item.

The solution, once you know the source of the problem, is easy: temporarily increment the reference count. The correct version of the function reads:

voidno\_bug(PyObject \*list){ PyObject \*item = PyList\_GetItem(list, 0);

Py\_INCREF(item); PyList\_SetItem(list, 1, PyLong\_FromLong(0L)); PyObject\_Print(item, stdout, 0); Py\_DECREF(item);}

This is a true story. An older version of Python contained variants of this bug and someone spent a considerable amount of time in a C debugger to figure out why his \_\_del\_\_() methods would fail…

The second case of problems with a borrowed reference is a variant involving threads. Normally, multiple threads in the Python interpreter can’t get in each other’s way, because there is a global lock protecting Python’s entire object space. However, it is possible to temporarily release this lock using the macro [Py\_BEGIN\_ALLOW\_THREADS](https://docs.python.org/3/c-api/init.html" \l "c.Py_BEGIN_ALLOW_THREADS" \o "Py_BEGIN_ALLOW_THREADS), and to re-acquire it using [Py\_END\_ALLOW\_THREADS](https://docs.python.org/3/c-api/init.html" \l "c.Py_END_ALLOW_THREADS" \o "Py_END_ALLOW_THREADS). This is common around blocking I/O calls, to let other threads use the processor while waiting for the I/O to complete. Obviously, the following function has the same problem as the previous one:

voidbug(PyObject \*list){ PyObject \*item = PyList\_GetItem(list, 0); Py\_BEGIN\_ALLOW\_THREADS ...some blocking I/O call... Py\_END\_ALLOW\_THREADS PyObject\_Print(item, stdout, 0); */\* BUG! \*/*}

### 1.10.4. NULL Pointers

In general, functions that take object references as arguments do not expect you to pass them NULL pointers, and will dump core (or cause later core dumps) if you do so. Functions that return object references generally return NULL only to indicate that an exception occurred. The reason for not testing for NULL arguments is that functions often pass the objects they receive on to other function — if each function were to test for NULL, there would be a lot of redundant tests and the code would run more slowly.

It is better to test for NULL only at the “source:” when a pointer that may be NULL is received, for example, from malloc() or from a function that may raise an exception.

The macros [Py\_INCREF()](https://docs.python.org/3/c-api/refcounting.html" \l "c.Py_INCREF" \o "Py_INCREF) and [Py\_DECREF()](https://docs.python.org/3/c-api/refcounting.html" \l "c.Py_DECREF" \o "Py_DECREF) do not check for NULL pointers — however, their variants [Py\_XINCREF()](https://docs.python.org/3/c-api/refcounting.html" \l "c.Py_XINCREF" \o "Py_XINCREF) and [Py\_XDECREF()](https://docs.python.org/3/c-api/refcounting.html" \l "c.Py_XDECREF" \o "Py_XDECREF) do.

The macros for checking for a particular object type (Pytype\_Check()) don’t check for NULL pointers — again, there is much code that calls several of these in a row to test an object against various different expected types, and this would generate redundant tests. There are no variants with NULL checking.

The C function calling mechanism guarantees that the argument list passed to C functions (args in the examples) is never NULL — in fact it guarantees that it is always a tuple [4](https://docs.python.org/3/extending/extending.html" \l "id8).

It is a severe error to ever let a NULL pointer “escape” to the Python user.

9.

* **[2أنماط التصميم الإنشائية Creational Design Patterns](https://wiki.hsoub.com/Design_Patterns" \l ".D8.A3.D9.86.D9.85.D8.A7.D8.B7_.D8.A7.D9.84.D8.AA.D8.B5.D9.85.D9.8A.D9.85_.D8.A7.D9.84.D8.A5.D9.86.D8.B4.D8.A7.D8.A6.D9.8A.D8.A9_Creational_Design_Patterns)**
  + **[2.1نمط أسلوب المصنع Factory Method](https://wiki.hsoub.com/Design_Patterns" \l ".D9.86.D9.85.D8.B7_.D8.A3.D8.B3.D9.84.D9.88.D8.A8_.D8.A7.D9.84.D9.85.D8.B5.D9.86.D8.B9_Factory_Method)**
  + **[2.2نمط المصنع المجرد Abstract Factory](https://wiki.hsoub.com/Design_Patterns" \l ".D9.86.D9.85.D8.B7_.D8.A7.D9.84.D9.85.D8.B5.D9.86.D8.B9_.D8.A7.D9.84.D9.85.D8.AC.D8.B1.D8.AF_Abstract_Factory)**
  + **[2.3نمط الباني Builder](https://wiki.hsoub.com/Design_Patterns" \l ".D9.86.D9.85.D8.B7_.D8.A7.D9.84.D8.A8.D8.A7.D9.86.D9.8A_Builder)**
  + **[2.4نمط النموذج الأولي Prototype](https://wiki.hsoub.com/Design_Patterns" \l ".D9.86.D9.85.D8.B7_.D8.A7.D9.84.D9.86.D9.85.D9.88.D8.B0.D8.AC_.D8.A7.D9.84.D8.A3.D9.88.D9.84.D9.8A_Prototype)**
  + **[2.5نمط المفردة Singleton](https://wiki.hsoub.com/Design_Patterns" \l ".D9.86.D9.85.D8.B7_.D8.A7.D9.84.D9.85.D9.81.D8.B1.D8.AF.D8.A9_Singleton)**
* **[3أنماط التصميم الهيكلية](https://wiki.hsoub.com/Design_Patterns" \l ".D8.A3.D9.86.D9.85.D8.A7.D8.B7_.D8.A7.D9.84.D8.AA.D8.B5.D9.85.D9.8A.D9.85_.D8.A7.D9.84.D9.87.D9.8A.D9.83.D9.84.D9.8A.D8.A9)**
  + **[3.1نمط المحوِّل Adapter](https://wiki.hsoub.com/Design_Patterns" \l ".D9.86.D9.85.D8.B7_.D8.A7.D9.84.D9.85.D8.AD.D9.88.D9.90.D9.91.D9.84_Adapter)**
  + **[3.2نمط الجسر Bridge](https://wiki.hsoub.com/Design_Patterns" \l ".D9.86.D9.85.D8.B7_.D8.A7.D9.84.D8.AC.D8.B3.D8.B1_Bridge)**
  + **[3.3نمط المُركَّب Composite](https://wiki.hsoub.com/Design_Patterns" \l ".D9.86.D9.85.D8.B7_.D8.A7.D9.84.D9.85.D9.8F.D8.B1.D9.83.D9.8E.D9.91.D8.A8_Composite)**
  + **[3.4نمط المُزخرِف Decorator](https://wiki.hsoub.com/Design_Patterns" \l ".D9.86.D9.85.D8.B7_.D8.A7.D9.84.D9.85.D9.8F.D8.B2.D8.AE.D8.B1.D9.90.D9.81_Decorator)**
  + **[3.5نمط الواجهة Facade](https://wiki.hsoub.com/Design_Patterns" \l ".D9.86.D9.85.D8.B7_.D8.A7.D9.84.D9.88.D8.A7.D8.AC.D9.87.D8.A9_Facade)**
  + **[3.6نمط وزن الذبابة Flyweight](https://wiki.hsoub.com/Design_Patterns" \l ".D9.86.D9.85.D8.B7_.D9.88.D8.B2.D9.86_.D8.A7.D9.84.D8.B0.D8.A8.D8.A7.D8.A8.D8.A9_Flyweight)**
  + **[3.7نمط الوكيل Proxy](https://wiki.hsoub.com/Design_Patterns" \l ".D9.86.D9.85.D8.B7_.D8.A7.D9.84.D9.88.D9.83.D9.8A.D9.84_Proxy)**
* **[4أنماط التصميم السلوكية](https://wiki.hsoub.com/Design_Patterns" \l ".D8.A3.D9.86.D9.85.D8.A7.D8.B7_.D8.A7.D9.84.D8.AA.D8.B5.D9.85.D9.8A.D9.85_.D8.A7.D9.84.D8.B3.D9.84.D9.88.D9.83.D9.8A.D8.A9)**
  + **[4.1نمط سلسلة المسؤولية Chain of Responsibility](https://wiki.hsoub.com/Design_Patterns" \l ".D9.86.D9.85.D8.B7_.D8.B3.D9.84.D8.B3.D9.84.D8.A9_.D8.A7.D9.84.D9.85.D8.B3.D8.A4.D9.88.D9.84.D9.8A.D8.A9_Chain_of_Responsibility)**
  + **[4.2نمط الأمر Command](https://wiki.hsoub.com/Design_Patterns" \l ".D9.86.D9.85.D8.B7_.D8.A7.D9.84.D8.A3.D9.85.D8.B1_Command)**
  + **[4.3نمط المكرِّر Iterator](https://wiki.hsoub.com/Design_Patterns" \l ".D9.86.D9.85.D8.B7_.D8.A7.D9.84.D9.85.D9.83.D8.B1.D9.90.D9.91.D8.B1_Iterator)**
  + **[4.4نمط الوسيط Mediator](https://wiki.hsoub.com/Design_Patterns" \l ".D9.86.D9.85.D8.B7_.D8.A7.D9.84.D9.88.D8.B3.D9.8A.D8.B7_Mediator)**
  + **[4.5نمط التذكرة Memento](https://wiki.hsoub.com/Design_Patterns" \l ".D9.86.D9.85.D8.B7_.D8.A7.D9.84.D8.AA.D8.B0.D9.83.D8.B1.D8.A9_Memento)**
  + **[4.6نمط المراقِب Observer](https://wiki.hsoub.com/Design_Patterns" \l ".D9.86.D9.85.D8.B7_.D8.A7.D9.84.D9.85.D8.B1.D8.A7.D9.82.D9.90.D8.A8_Observer)**
  + **[4.7نمط الحالة State](https://wiki.hsoub.com/Design_Patterns" \l ".D9.86.D9.85.D8.B7_.D8.A7.D9.84.D8.AD.D8.A7.D9.84.D8.A9_State)**
  + **[4.8نمط الخطة Strategy](https://wiki.hsoub.com/Design_Patterns" \l ".D9.86.D9.85.D8.B7_.D8.A7.D9.84.D8.AE.D8.B7.D8.A9_Strategy)**
  + **[4.9نمط أسلوب القالب Template Method](https://wiki.hsoub.com/Design_Patterns" \l ".D9.86.D9.85.D8.B7_.D8.A3.D8.B3.D9.84.D9.88.D8.A8_.D8.A7.D9.84.D9.82.D8.A7.D9.84.D8.A8_Template_Method)**

**[4.10نمط الزائر Visitor](https://wiki.hsoub.com/Design_Patterns" \l ".D9.86.D9.85.D8.B7_.D8.A7.D9.84.D8.B2.D8.A7.D8.A6.D8.B1_Visitor)**

**10.**

### قواعد البيانات اليدوية  Manual Database

1. 

تلك البيانات المبعثرة في الأرفف والأوراق التي تملأ المكتبات والمخازن تمثل قواعد البيانات اليدوية . و بسبب هذه البعثرة جاءت تصنيفات مهمة ومنطقية لتصنف البيانات حسب علاقتها ببعضها أو حسب بنيتها الرياضية او المنطقية، و بناء على ما ذكرتُ لك تُعتبر الملفات المخزنة في الأرفف و فواتير المبيعات المجدولة إلكترونياً و عناوين الأشخاص في ملف وثائقي أمثلةً لبيانات يدوية.

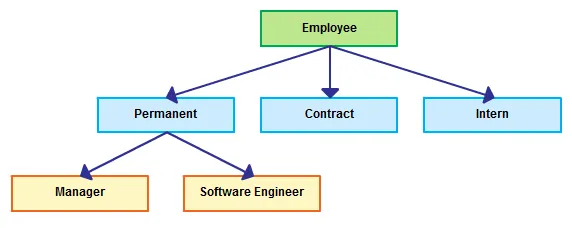
### قواعد البيانات العلائقية Relational Database

هذا النوع من قواعد البيانات من أكثر الأنواع إستخداماً من قبل الإنسان من أجل تنسيق المعلومات، فهو يعتمد على ربط الجداول والمعلومات بطريقة أسهل من أجل سرعة الوصول إلى المعلومات المطلوبة. و هي من الطرق التي يستخدمها العقل البشري كثيراً في محاولات التذكر للأحداث القديمة. علاقة بسيطة قد تجعلك تتذكر أحداثاً كبيرة.  
يمتلك هذا النوع من قواعد البيانات المعلومات مميزات جيدة، كأن يتم إدخالها البيانات مرة واحدة فقط فلا داعي للتكرار. كما أن الجداول الصغيرة يمكن إنشائها وتعديلها بسهولة. فالصغير الواضح ذو العلاقات البينة أسهل في التعديل، إضافة إلى إمكانية إضافة الجداول إلى قاعدة البيانات في أي وقت.

### قواعد البيانات غير العلائقية Non-relational Database

وفى هذا النظام يُنشأ جدول كبير يحتوى على جميع البيانات. كأن كل ما تملك من معلومات في ورقة وحيدة.  
لا يخفى عليك غرابة هذه الطريقة و ربما تخيلت عدة مساوئ لها، ففي هذا النوع من قواعد البيانات تتكرر البيانات بكثرة ، ففي حالة إدخالك لمنتج 10 مرات فسيكتب رقم هاتف المورد مثلاً 10 مرات كذلك!! ولا يخفي عليك عند التعديل سيُعدل الرقم كذل 10 مرات أيضاً!!

### قواعد بيانات ذات الشكل هرمي Hierarchy Database

1. 

وتعتمد هذه القاعدة على مبدأ التسلسل الهرمي في العمل . حيث أنها تقوم بعمل تسلسل من الأصل ، أو الجذر ، حيث أن هذا النظام يبدأ في التفرع على شكل أقسام ، ويقوم مبدأ عمله على الوصول إلى البيانات بطريقة متسلسلة ومتفرعة ، وتكون إما من أسفل للأعلى أو من الأعلى للأسفل .

### قواعد البيانات الشبكية Network Database

ظهر هذا النوع من قواعد البيانات في زمن شهرة قواعد البيانات ذات الشكل الهرمي. يعتمد الشكل الهرمي على أن يكون الأب وحيد و له عدة أبناء، و لكن وُجد أن بعض البيانات ترتبط بطريقة عدة أبناء مع عدة آباء و العكس صحيح. ربما ليس من المنطقي تعدد الآباء في الحقيقه لكنه في قواعد البيانات يحدُث  .  
يقتصر النوعان الرابع والخامس على الإحتياجات الكبيرة لأنهما يتطلبان عادة ذواكر بأحجام كبيرة. ولكن رغم ذلك فإنها لها مزايا عديدة، فهي أكثر كفاءة من قواعد البيانات العلائقية ، وتتعامل مع كم كبير جداً من المعلومات ، بإضافة إلى توفير بناء على طريقة تنظيم الملفات التي تتبعها مساحات كبيرة من وسائط لتخزين البيانات Storage Data base.



**الفرق بين الهرمية والشبكية والعلائقية :**يستخدم النموذجان الهرمي والشبكي روابط (**links**) أو مؤشرات (**pointers**) لوصل السجلات (**Records**) ببعضها البعض في النظام ، وتدعى هذه الأنظمة بالأنظمة الساكنة (**static**) أو المتراصة (**monolithic**) لأن السجلات فيها مربوطة ببعضها بشكل فيزيائي من خلال تعاريفها، وتتميز هذه الأنظمة بأنها معقدة العمل وصعبة التعديل، إلا أن سرعة الوصول فيها تغطي عيوبها .

أما في الأنظمة العلائقية فالربط بين السجلات لا يجري فيزيائياً عن طريق المؤشرات، وإنما عن طريق الأسماء الحقيقية للحقول ، كحقول رقم الموظف أو الإسم أو رقم البطاقة، فالسجلات في هذا النظام قابلة للعنونة بالمحتوى (**connect-addressable**) بحيث يجري الوصول إليها بمطابقة قيم البيانات المخزنة مع بعضها.

تضم الأسواق حالياً مالا يقل عن 200 نظام (**Data Base Managements Systems DBMS**) لإدارة قواعد البيانات ، نصفها تقريباً يستخدم اللغة **SQL** (أنظمة علائقية) على مختلف أنواع الحاسبات.

1. 

تلك البيانات المبعثرة في الأرفف والأوراق التي تملأ المكتبات والمخازن تمثل قواعد البيانات اليدوية . و بسبب هذه البعثرة جاءت تصنيفات مهمة ومنطقية لتصنف البيانات حسب علاقتها ببعضها أو حسب بنيتها الرياضية او المنطقية، و بناء على ما ذكرتُ لك تُعتبر الملفات المخزنة في الأرفف و فواتير المبيعات المجدولة إلكترونياً و عناوين الأشخاص في ملف وثائقي أمثلةً لبيانات يدوية.

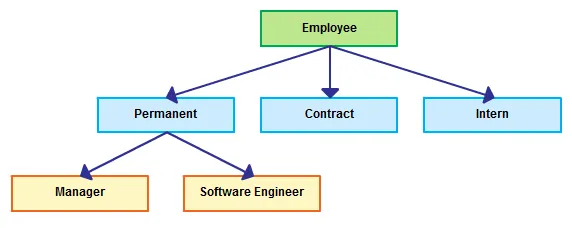
### قواعد البيانات العلائقية Relational Database

هذا النوع من قواعد البيانات من أكثر الأنواع إستخداماً من قبل الإنسان من أجل تنسيق المعلومات، فهو يعتمد على ربط الجداول والمعلومات بطريقة أسهل من أجل سرعة الوصول إلى المعلومات المطلوبة. و هي من الطرق التي يستخدمها العقل البشري كثيراً في محاولات التذكر للأحداث القديمة. علاقة بسيطة قد تجعلك تتذكر أحداثاً كبيرة.  
يمتلك هذا النوع من قواعد البيانات المعلومات مميزات جيدة، كأن يتم إدخالها البيانات مرة واحدة فقط فلا داعي للتكرار. كما أن الجداول الصغيرة يمكن إنشائها وتعديلها بسهولة. فالصغير الواضح ذو العلاقات البينة أسهل في التعديل، إضافة إلى إمكانية إضافة الجداول إلى قاعدة البيانات في أي وقت.

### قواعد البيانات غير العلائقية Non-relational Database

وفى هذا النظام يُنشأ جدول كبير يحتوى على جميع البيانات. كأن كل ما تملك من معلومات في ورقة وحيدة.  
لا يخفى عليك غرابة هذه الطريقة و ربما تخيلت عدة مساوئ لها، ففي هذا النوع من قواعد البيانات تتكرر البيانات بكثرة ، ففي حالة إدخالك لمنتج 10 مرات فسيكتب رقم هاتف المورد مثلاً 10 مرات كذلك!! ولا يخفي عليك عند التعديل سيُعدل الرقم كذل 10 مرات أيضاً!!

### قواعد بيانات ذات الشكل هرمي Hierarchy Database

1. 

وتعتمد هذه القاعدة على مبدأ التسلسل الهرمي في العمل . حيث أنها تقوم بعمل تسلسل من الأصل ، أو الجذر ، حيث أن هذا النظام يبدأ في التفرع على شكل أقسام ، ويقوم مبدأ عمله على الوصول إلى البيانات بطريقة متسلسلة ومتفرعة ، وتكون إما من أسفل للأعلى أو من الأعلى للأسفل .

### قواعد البيانات الشبكية Network Database

ظهر هذا النوع من قواعد البيانات في زمن شهرة قواعد البيانات ذات الشكل الهرمي. يعتمد الشكل الهرمي على أن يكون الأب وحيد و له عدة أبناء، و لكن وُجد أن بعض البيانات ترتبط بطريقة عدة أبناء مع عدة آباء و العكس صحيح. ربما ليس من المنطقي تعدد الآباء في الحقيقه لكنه في قواعد البيانات يحدُث  .  
يقتصر النوعان الرابع والخامس على الإحتياجات الكبيرة لأنهما يتطلبان عادة ذواكر بأحجام كبيرة. ولكن رغم ذلك فإنها لها مزايا عديدة، فهي أكثر كفاءة من قواعد البيانات العلائقية ، وتتعامل مع كم كبير جداً من المعلومات ، بإضافة إلى توفير بناء على طريقة تنظيم الملفات التي تتبعها مساحات كبيرة من وسائط لتخزين البيانات Storage Data base.



**الفرق بين الهرمية والشبكية والعلائقية :**يستخدم النموذجان الهرمي والشبكي روابط (**links**) أو مؤشرات (**pointers**) لوصل السجلات (**Records**) ببعضها البعض في النظام ، وتدعى هذه الأنظمة بالأنظمة الساكنة (**static**) أو المتراصة (**monolithic**) لأن السجلات فيها مربوطة ببعضها بشكل فيزيائي من خلال تعاريفها، وتتميز هذه الأنظمة بأنها معقدة العمل وصعبة التعديل، إلا أن سرعة الوصول فيها تغطي عيوبها .

أما في الأنظمة العلائقية فالربط بين السجلات لا يجري فيزيائياً عن طريق المؤشرات، وإنما عن طريق الأسماء الحقيقية للحقول ، كحقول رقم الموظف أو الإسم أو رقم البطاقة، فالسجلات في هذا النظام قابلة للعنونة بالمحتوى (**connect-addressable**) بحيث يجري الوصول إليها بمطابقة قيم البيانات المخزنة مع بعضها.

تضم الأسواق حالياً مالا يقل عن 200 نظام (**Data Base Managements Systems DBMS**) لإدارة قواعد البيانات ، نصفها تقريباً يستخدم اللغة **SQL** (أنظمة علائقية) على مختلف أنواع الحاسبات.