Chapter 14. Customizing Execution

Python exposes, supports, and documents many of its internal mechanisms. This may help you understand Python at an advanced level, and lets you hook your own code into such Python mechanisms, controlling them to some extent. For example, **"Python built-ins"** covers the way Python arranges for built-ins to be visible. This chapter covers some other advanced Python techniques, including site customization, termination functions, dynamic execution, handling internal types, and garbage collection. We'll look at other issues related to controlling execution using multiple threads and processes in **Chapter 15**; **Chapter 17** covers issues specific to testing, debugging, and profiling.

Per-Site Customization

Python provides a specific "hook" to let each site customize some aspects of Python's behavior at the start of each run. Python loads the standard module `site` just before the main script. If Python is run with the option **`-S`**, it does not load `site`. **`-S`** allows faster startup but saddles the main script with initialization chores. `site`'s tasks are, chiefly, to put `sys.path` in standard form (absolute paths, no duplicates), including as directed by environment variables, by virtual environments, and by each *.pth* file found in a directory in `sys.path`.

Secondarily, if the session starting is an interactive one, `site` adds several handy built-ins (such as `exit`, `copyright`, etc.) and, if `readline` is enabled, configure autocompletion as the function of the Tab key.

In any normal Python installation, the installation process sets everything up to ensure that `site`'s work is sufficient to let Python programs and interactive sessions run "normally," i.e., as they would on any other system

with that version of Python installed. In exceptional cases, if as a system administrator (or in an equivalent role, such as a user who has installed Python in their home directory for their sole use) you think you absolutely need to do some customization, perform it in a new file called *sitecustomize.py* (create it in the same directory where *site.py* lives).

AVOID MODIFYING SITE.PY

We strongly recommend that you do *not* alter the *site.py* file that performs the base customization. Doing so might cause Python to behave differently on your system than elsewhere. In any case, the *site.py* file will be overwritten each and every time you update your Python installation, and your modifications will be lost.

In the rare cases where *sitecustomize.py* is present, what it typically does is add yet more dictionaries to `sys.path`—the best way to perform this task is for *sitecustomize.py* to **`import`** `site` and then to call `site.addsitedir(`*`path_to_a_dir`*`)`.

Termination Functions

The `atexit` module lets you register termination functions (i.e., functions to be called at program termination, in LIFO order). Termination functions are similar to cleanup handlers established by **`try/finally`** or **`with`**. However, termination functions are globally registered and get called at the end of the whole program, while cleanup handlers are established lexically and get called at the end of a specific **`try`** clause or **`with`** statement. Termination functions and cleanup handlers are called whether the program terminates normally or abnormally, but not when the program ends by calling `os._exit` (so you normally call `sys.exit` instead). The `atexit` module supplies a function called `register` that takes as arguments *`func`*, *`*args`*, and *`*kwds`* and ensures that *`func`*`(*`*`args`*`, **`*`kwds`*`)` is called at program termination time.

Dynamic Execution and exec

Python's exec built-in function can execute code that you read, generate, or otherwise obtain during a program's run. exec dynamically executes a statement or a suite of statements. It has the following syntax:

```
exec(code, globals=None, locals=None, /)
```

code can be a str, bytes, bytearray, or code object. *globals* is a dict, and *locals* can be any mapping.

If you pass both *globals* and *locals*, they are the global and local namespaces in which *code* runs. If you pass only *globals*, exec uses *globals* as both namespaces. If you pass neither, *code* runs in the current scope.

NEVER RUN EXEC IN THE CURRENT SCOPE

Running exec in the current scope is a particularly bad idea: it can bind, rebind, or unbind any global name. To keep things under control, use exec, if at all, only with specific, explicit dictionaries.

Avoiding exec

A frequently asked question about Python is "How do I set a variable whose name I just read or built?" Literally, for a *global* variable, exec allows this, but it's a very bad idea to use exec for this purpose. For example, if the name of the variable is in *varname*, you might think to use:

```
exec(varname + ' = 23')
```

Don't do this. An exec like this in the current scope makes you lose control of your namespace, leading to bugs that are extremely hard to find and making your program unfathomably difficult to understand. Keep the

"variables" that you need to set with dynamically found names not as actual variables, but as entries in a dictionary (say, *mydict*). You might then consider using:

```
exec(varname+'=23', mydict) # Still a bad idea
```

While this is not quite as terrible as the previous example, it is still a bad idea. Keeping such "variables" as dictionary entries means that you don't have any need to use exec to set them! Just code:

```
mydict[varname] = 23
```

This way, your program is clearer, direct, elegant, and faster. There *are* some valid uses of exec, but they are extremely rare: just use explicit dictionaries instead.

STRIVE TO AVOID EXEC

Use exec only when it's really indispensable, which is *extremely* rare. Most often, it's best to avoid exec and choose more specific, well-controlled mechanisms: exec weakens your control of your code's namespace, can damage your program's performance, and exposes you to numerous hard-to-find bugs and huge security risks.

Expressions

exec can execute an expression, because any expression is also a valid statement (called an *expression statement*). However, Python ignores the value returned by an expression statement. To evaluate an expression and obtain the expression's value, use the built-in function eval, covered in **Table 8-2**. (Note, however, that just about all of the same security risk caveats for exec also apply to eval.)

compile and Code Objects

To make a code object to use with `exec`, call the built-in function `compile` with the last argument set to `'exec'` (as covered in **Table 8-2**).

A code object *c* exposes many interesting read-only attributes whose names all start with `'co_'`, such as those listed in **Table 14-1**.

Table 14-1. Read-only attributes of code objects

`co_argcount`	The number of parameters of the function of which *c* is the code (`0` when *c* is not the code object of a function, but rather is built directly by `compile`)
`co_code`	A `bytes` object with *c*'s bytecode
`co_consts`	The tuple of constants used in *c*
`co_filename`	The name of the file *c* was compiled from (the string that is the second argument to `compile`, when *c* was built that way)
`co_firstlineno`	The initial line number (within the file named by `co_filename`) of the source code that was compiled to produce *c*, if *c* was built by compiling from a file
`co_name`	The name of the function of which *c* is the code (`'<module>'` when *c* is not the code object of a function but rather is built directly by `compile`)
`co_names`	The tuple of all identifiers used within *c*
`co_varnames`	The tuple of local variables' identifiers in *c*, starting with parameter names

Most of these attributes are useful only for debugging purposes, but some may help with advanced introspection, as exemplified later in this

section.

If you start with a string holding one or more statements, first use `compile` on the string, then call `exec` on the resulting code object—that's a bit better than giving `exec` the string to compile and execute. This separation lets you check for syntax errors separately from execution-time errors. You can often arrange things so that the string is compiled once and the code object executes repeatedly, which speeds things up. `eval` can also benefit from such separation. Moreover, the `compile` step is intrinsically safe (both `exec` and `eval` are extremely risky if you execute them on code that you don't 100% trust), and you may be able to check the code object before it executes, to lessen the risk (though it will never truly be zero).

As mentioned in **Table 14-1**, a code object has a read-only attribute `co_names` that is the tuple of the names used in the code. For example, say that you want the user to enter an expression that contains only literal constants and operators—no function calls or other names. Before evaluating the expression, you can check that the string the user entered satisfies these constraints:

```
def safer_eval(s):
    code = compile(s, '<user-entered string>', 'eval')
    if code.co_names:
        raise ValueError(
            f'Names {code.co_names!r} not allowed in expression {s!r}')
    return eval(code)
```

This function `safer_eval` evaluates the expression passed in as argument `s` only when the string is a syntactically valid expression (otherwise, `compile` raises `SyntaxError`) and contains no names at all (otherwise, `safer_eval` explicitly raises `ValueError`). (This is similar to the standard library function `ast.literal_eval`, covered in **"Standard Input"**, but a bit more powerful, since it does allow the use of operators.)

Knowing what names the code is about to access may sometimes help you optimize the preparation of the dictionary that you need to pass to `exec`

or `eval` as the namespace. Since you need to provide values only for those names, you may save work by not preparing other entries. For example, say that your application dynamically accepts code from the user, with the convention that variable names starting with `data_` refer to files residing in the subdirectory *data* that user-written code doesn't need to read explicitly. User-written code may, in turn, compute and leave results in global variables with names starting with `result_`, which your application writes back as files in the *data* subdirectory. Thanks to this convention, you may later move the data elsewhere (e.g., to BLOBs in a database instead of files in a subdirectory), and user-written code won't be affected. Here's how you might implement these conventions efficiently:

```
def exec_with_data(user_code_string):
    user_code = compile(user_code_string, '<user code>', 'exec')
    datadict = {}
    for name in user_code.co_names:
        if name.startswith('data_'):
            with open(f'data/{name[5:]}', 'rb') as datafile:
                datadict[name] = datafile.read()
        elif name.startswith('result_'):
            pass  # user code assigns to variables named `result_...`
        else:
            raise ValueError(f'invalid variable name {name!r}')
    exec(user_code, datadict)
    for name in datadict:
        if name.startswith('result_'):
            with open(f'data/{name[7:]}', 'wb') as datafile:
                datafile.write(datadict[name])
```

Never exec or eval Untrusted Code

Some older versions of Python supplied tools that aimed to ameliorate the risks of using `exec` and `eval`, under the heading of "restricted execution," but those tools were never entirely secure against the ingenuity of able hackers, and recent versions of Python have dropped them to avoid offering the user an unfounded sense of security. If you need to guard against such attacks, take advantage of your operating system's protection mech-

anisms: run untrusted code in a separate process, with privileges as restricted as you can possibly make them (study the mechanisms that your OS supplies for the purpose, such as `chroot`, `setuid`, and `jail`; in Windows, you might try the third-party commercial add-on **WinJail**, or run untrusted code in a separate, highly constrained virtual machine or container, if you're an expert on how to securitize containers). To guard against denial of service attacks, have the main process monitor the separate one and terminate the latter if and when resource consumption becomes excessive. Processes are covered in **"Running Other Programs"**.

EXEC AND EVAL ARE UNSAFE WITH UNTRUSTED CODE

The function `exec_with_data` defined in the previous section is not at all safe against untrusted code: if you pass to it, as the argument *`user_code`*, some string obtained in a way that you cannot *entirely* trust, there is essentially no limit to the amount of damage it might do. This is unfortunately true of just about any use of `exec` or `eval`, except for those rare cases in which you can set extremely strict and fully checkable limits on the code to execute or evaluate, as was the case for the function `safer_eval`.

Internal Types

Some of the internal Python objects described in this section are hard to use, and indeed are not meant for you to use most of the time. Using such objects correctly and to good effect requires some study of your Python implementation's C sources. Such black magic is rarely needed, except for building general-purpose development tools and similar wizardly tasks. Once you do understand things in depth, Python empowers you to exert control if and when needed. Since Python exposes many kinds of internal objects to your Python code, you can exert that control by coding in Python, even when you need an understanding of C to read Python's sources and understand what's going on.

Type Objects

The built-in type named `type` acts as a callable factory, returning objects that are types. Type objects don't have to support any special operations except equality comparison and representation as strings. However, most type objects are callable and return new instances of the type when called. In particular, built-in types such as `int`, `float`, `list`, `str`, `tuple`, `set`, and `dict` all work this way; specifically, when called without arguments, they return a new empty instance, or, for numbers, one that equals `0`. The attributes of the `types` module are the built-in types that don't have built-in names. Besides being callable to generate instances, type objects are useful because you can inherit from them, as covered in **"Classes and Instances"**.

The Code Object Type

Besides using the built-in function `compile`, you can get a code object via the `__code__` attribute of a function or method object. (For a discussion of the attributes of code objects, see **"compile and Code Objects"**.) Code objects are not callable, but you can rebind the `__code__` attribute of a function object with the right number of parameters in order to wrap a code object into callable form. For example:

```
def g(x):
    print('g', x)
code_object = g.__code__
def f(x):
    pass
f.__code__ = code_object
f(23)     # prints: g 23
```

Code objects that have no parameters can also be used with `exec` or `eval`. Directly creating code objects requires many parameters; see Stack Overflow's **unofficial docs** on how to do it (but bear in mind that you're usually better off calling `compile` instead).

The Frame Type

The function `_getframe` in the module `sys` returns a frame object from Python's call stack. A frame object has attributes giving information about the code executing in the frame and the execution state. The `traceback` and `inspect` modules help you access and display such information, particularly when an exception is being handled. **Chapter 17** provides more information about frames and tracebacks, and covers the module `inspect`, which is the best way to perform such introspection. As the leading underscore in the name `_getframe` hints, the function is "somewhat private"; it's meant for use only by tools such as debuggers, which inevitably require deep introspection into Python's internals.

Garbage Collection

Python's garbage collection normally proceeds transparently and automatically, but you can choose to exert some direct control. The general principle is that Python collects each object *x* at some time after *x* becomes unreachable—that is, when no chain of references can reach *x* by starting from a local variable of a function instance that is executing, or from a global variable of a loaded module. Normally, an object *x* becomes unreachable when there are no references at all to *x*. In addition, a group of objects can be unreachable when they reference each other but no global or local variables reference any of them, even indirectly (such a situation is known as a *mutual reference loop*).

Classic Python keeps with each object *x* a count, known as a *reference count*, of how many references to *x* are outstanding. When *x*'s reference count drops to `0`, CPython immediately collects *x*. The function `getrefcount` of the module `sys` accepts any object and returns its reference count (at least `1`, since `getrefcount` itself has a reference to the object it's examining). Other versions of Python, such as Jython or PyPy, rely on other garbage collection mechanisms supplied by the platform they run on (e.g., the JVM or the LLVM). The modules `gc` and `weakref`, therefore, apply only to CPython.

When Python garbage-collects *x* and there are no references to *x*, Python finalizes *x* (i.e., calls *x*`.__del__`) and frees the memory that *x* occupied. If *x* held any references to other objects, Python removes the references, which in turn may make other objects collectable by leaving them unreachable.

The gc Module

The `gc` module exposes the functionality of Python's garbage collector. `gc` deals with unreachable objects that are part of mutual reference loops. As mentioned previously, in such a loop, each object in the loop refers to one or more of the others, keeping the reference counts of all the objects positive, but there are no outside references to any of the set of mutually referencing objects. Therefore, the whole group, also known as *cyclic garbage*, is unreachable and thus garbage-collectable. Looking for such cyclic garbage loops takes time, which is why the module `gc` exists: to help you control whether and when your program spends that time. By default, cyclic garbage collection functionality is enabled with some reasonable default parameters: however, by importing the `gc` module and calling its functions, you may choose to disable the functionality, change its parameters, and/or find out exactly what's going on in this respect.

`gc` exposes attributes and functions to help you manage and instrument cyclic garbage collection, including those listed in **Table 14-2**. These functions can let you track down memory leaks—objects that are not collected even though there *should* be no more references to them—by helping you discover what other objects are in fact holding on to references to them. Note that `gc` implements the architecture known in computer science as **generational garbage collection**.

Table 14-2. `gc` functions and attributes

`callbacks`	A `list` of callbacks that the garbage collector will invoke before and after collection. See **"Instrumenting garbage collection"** for further details.

`collect`	`collect()` Forces a full cyclic garbage collection run to happen immediately.
`disable`	`disable()` Suspends automatic, periodic cyclic garbage collection.
`enable`	`enable()` Reenables periodic cyclic garbage collection previously suspended with `disable`.
`freeze`	`freeze()` Freezes all objects tracked by `gc`: moves them to a “permanent generation,” i.e., a set of objects to be ignored in all the future collections.
`garbage`	A `list` (but, treat it as read-only) of unreachable but uncollectable objects. This happens when any object in a cyclic garbage loop has a `__del__` special method, as there may be no demonstrably safe order for Python to finalize such objects.
`get_count`	`get_count()` Returns the current collection counts as a tuple, `(count0, count1, count2)`.
`get_debug`	`get_debug()` Returns an `int` bit string, the garbage collection debug flags set with `set_debug`.
`get_freeze_count`	`get_freeze_count()` Returns the number of objects in the permanent generation.

`get_objects`	`get_objects(generation=None)` Returns a list of objects being tracked by the collector. **3.8+** If the optional `generation` argument is not **`None`**, lists only those objects in the selected generation.
`get_referents`	`get_referents(*objs)` Returns a list of objects, visited by the arguments' C-level `tp_traverse` methods, that are referred to by any of the arguments.
`get_referrers`	`get_referrers(*objs)` Returns a list of all container objects currently tracked by the cyclic garbage collector that refer to any one or more of the arguments.
`get_stats`	`get_stats()` Returns a list of three `dicts`, one per generation, containing counts of the number of collections, the number of objects collected, and the number of uncollectable objects.
`get_threshold`	`get_threshold()` Returns the current collection thresholds as a tuple of the three `ints`.
`isenabled`	`isenabled()` Returns **`True`** when cyclic garbage collection is currently enabled; otherwise returns **`False`.**
`is_finalized`	`is_finalized(obj)` **3.9+** Returns **`True`** when the garbage collector has finalized *`obj`*; otherwise, returns **`False`**.
`is_tracked`	`is_tracked(obj)` Returns **`True`** when *`obj`* is currently tracked by

	the garbage collector; otherwise, returns **`False`**.

`set_debug`	`set_debug(`*`flags`*`)`
	Sets flags for debugging behavior during garbage collection. *`flags`* is an `int`, interpreted as a bit string, built by ORing (with the bitwise OR operator, `|`) zero or more constants supplied by the module `gc`. Each bit enables a specific debugging function:
	`DEBUG_COLLECTABLE`
	Prints information on collectable objects found during garbage collection.
	`DEBUG_LEAK`
	Combines behavior for `DEBUG_COLLECTABLE`, `DEBUG_UNCOLLECTABLE`, and `DEBUG_SAVEALL`. Together, these are the most common flags used to help you diagnose memory leaks.
	`DEBUG_SAVEALL`
	Saves all collectable objects to the list `gc.garbage` (where uncollectable ones are also always saved) to help you diagnose leaks.
	`DEBUG_STATS`
	Prints statistics gathered during garbage collection to help you tune the thresholds.
	`DEBUG_UNCOLLECTABLE`
	Prints information on uncollectable objects found during garbage collection.

`set_threshold`	`set_threshold(`*`thresh0`*`[, `*`thresh1`*`[, `*`thresh2`*`]])` Sets thresholds that control how often cyclic garbage collection cycles run. A *`thresh0`* of `0` disables garbage collection. Garbage collection

	is an advanced, specialized topic, and the details of the generational garbage collection approach used in Python (and consequently the detailed meanings of these thresholds) are beyond the scope of this book; see the **online docs** for details.
`unfreeze`	`unfreeze()` Unfreezes all objects in the permanent generation, moving them all back to the oldest generation.

When you know there are no cyclic garbage loops in your program, or when you can't afford the delay of cyclic garbage collection at some crucial time, suspend automatic garbage collection by calling `gc.disable()`. You can enable collection again later by calling `gc.enable()`. You can test whether automatic collection is currently enabled by calling `gc.isenabled()`, which returns **`True`** or **`False`**. To control *when* time is spent collecting, you can call `gc.collect()` to force a full cyclic collection run to happen immediately. To wrap some time-critical code:

```
import gc
gc_was_enabled = gc.isenabled()
if gc_was_enabled:
    gc.collect()
    gc.disable()
# insert some time-critical code here
if gc_was_enabled:
    gc.enable()
```

You may find this easier to use if implemented as a context manager:

```
import gc
import contextlib
```

```
@contextlib.contextmanager
def gc_disabled():
    gc_was_enabled = gc.isenabled()
    if gc_was_enabled:
        gc.collect()
        gc.disable()
    try:
        yield
    finally:
        if gc_was_enabled:
            gc.enable()

with gc_disabled():
    # ...insert some time-critical code here...
```

Other functionality in the module `gc` is more advanced and rarely used, and can be grouped into two areas. The functions `get_threshold` and `set_threshold` and debug flag `DEBUG_STATS` help you fine-tune garbage collection to optimize your program's performance. The rest of `gc`'s functionality can help you diagnose memory leaks in your program. While `gc` itself can automatically fix many leaks (as long as you avoid defining `__del__` in your classes, since the existence of `__del__` can block cyclic garbage collection), your program runs faster if it avoids creating cyclic garbage in the first place.

Instrumenting garbage collection

`gc.callbacks` is an initially empty list to which you can add functions `f(`*`phase`*`,` *`info`*`)` which Python is to call upon garbage collection. When Python calls each such function, *`phase`* is `'start'` or `'stop'` to mark the beginning or end of a collection, and *`info`* is a dictionary containing information about the generational collection used by CPython. You can add functions to this list, for example to gather statistics about garbage collection. See the **documentation** for more details.

The weakref Module

Careful design can often avoid reference loops. However, at times you need objects to know about each other, and avoiding mutual references would distort and complicate your design. For example, a container has references to its items, yet it can often be useful for an object to know about a container holding it. The result is a reference loop: due to the mutual references, the container and items keep each other alive, even when all other objects forget about them. Weak references solve this problem by allowing objects to reference others without keeping them alive.

A *weak reference* is a special object *w* that refers to some other object *x* without incrementing *x*'s reference count. When *x*'s reference count goes down to `0`, Python finalizes and collects *x*, then informs *w* of *x*'s demise. Weak reference *w* can now either disappear or get marked as invalid in a controlled way. At any time, a given *w* refers to either the same object *x* as when *w* was created, or to nothing at all; a weak reference is never retargeted. Not all types of objects support being the target *x* of a weak reference *w*, but classes, instances, and functions do.

The `weakref` module exposes functions and types to create and manage weak references, detailed in **Table 14-3**.

Table 14-3. Functions and classes of the `weakref` module

`getweakrefcount`	`getweakrefcount(`*x*`)` Returns `len(getweakrefs(`*x*`))`.
`getweakrefs`	`getweakrefs(`*x*`)` Returns a list of all weak references and proxies whose target is *x*.
`proxy`	`proxy(`*x*`[,` *f*`])` Returns a weak proxy *p* of type `ProxyType` (`CallableProxyType` when *x* is callable) with *x* as the target. Using *p* is just like using *x*, except that, when you use *p* after *x* has been deleted, Python

	raises `ReferenceError`. *p* is never hashable (you cannot use *p* as a dictionary key). When *f* is present, it must be callable with one argument, and is the finalization callback for *p* (i.e., right before finalizing *x*, Python calls *f*(*p*)). *f* executes right *after* *x* is no longer reachable from *p*.
`ref`	`ref(`*x*`[,` *f*`])` Returns a weak reference *w* of type `ReferenceType` with object *x* as the target. *w* is callable without arguments: calling *w*`()` returns *x* when *x* is still alive; otherwise, *w*`()` returns `None`. *w* is hashable when *x* is hashable. You can compare weak references for equality (`==`, `!=`), but not for order (`<`, `>`, `<=`, `>=`). Two weak references *x* and *y* are equal when their targets are alive and equal, or when *x* is *y*. When *f* is present, it must be callable with one argument and is the finalization callback for *w* (i.e., right before finalizing *x*, Python calls *f*(*w*)). *f* executes right *after* *x* is no longer reachable from *w*.
`WeakKey Dictionary`	`class WeakKeyDictionary(adict={})` A `WeakKeyDictionary` *d* is a mapping weakly referencing its keys. When the reference count of a key *k* in *d* goes to `0`, item *d*`[`*k*`]` disappears. `adict` is used to initialize the mapping.
`WeakSet`	class WeakSet(elements=[]) A `WeakSet` *s* is a set weakly referencing its content elements, initialized from `elements`. When the reference count of an element *e* in *s* goes to `0`, *e* disappears from *s*.

`WeakValue Dictionary`	`class WeakValueDictionary(adict={})` A `WeakValueDictionary` *d* is a mapping weakly referencing its values. When the reference count of a value *v* in *d* goes to `0`, all items of *d* such that *d*[*k*] **`is`** *v* disappear. `adict` is used to initialize the mapping.

`WeakKeyDictionary` lets you noninvasively associate additional data with some hashable objects, with no change to the objects. `WeakValueDictionary` lets you noninvasively record transient associations between objects, and build caches. In each case, use a weak mapping, rather than a `dict`, to ensure that an object that is otherwise garbage-collectable is not kept alive just by being used in a mapping. Similarly, a `WeakSet` provides the same weak containment functionality in place of a normal `set`.

A typical example is a class that keeps track of its instances, but does not keep them alive just to keep track of them:

```
import weakref
class Tracking:
    _instances_dict = weakref.WeakValueDictionary()

    def __init__(self):
        Tracking._instances_dict[id(self)] = self

    @classmethod
    def instances(cls):
        return cls._instances_dict.values()
```

When the `Tracking` instances are hashable, a similar class can be implemented using a `WeakSet` of the instances, or a `WeakKeyDictionary` with the instances as keys and **`None`** for the values.