shortcut

Your Short Cut to Knowledge

Advanced Python 3 Programming Techniques

Mark Summerfield

Addison-Wesley
Pearson Education
informit.com/aw

Developer's Library
informit.com/devlibrary

shortcut

Your Short Cut to Knowledge

Advanced Python 3 Programming Techniques

Mark Summerfield

Addison-Wesley
Pearson Education
informit.com/aw

Developer's
Library
informit.com/devlibrary

Advanced Python 3 Programming Techniques

Mark Summerfield

Addison-Wesley
Pearson Education
informit.com/aw

Developer's Library
informit.com/devlibrary

Visit us on the Web: informit.com/aw

ISBN-13: 978-0-321-63551-8
ISBN-10: 0-321-63551-5

First release, February 2009

Table of Contents

Chapter 1. What This Short Cut Covers

This short cut is taken from *Programming in Python 3: A CompleteIntroduction to the Python Language* (Addison-Wesley, 2009) and provides self-contained coverage of Python's advanced features. Most of the techniques covered are not needed every day, but in the right circumstances they can make a crucial difference, allowing us to write clean and straightforward code rather than having to resort to hacks and workarounds to achieve what we need. The short cut explains a range of procedural, object-oriented, and functional-style techniques, and the information provided will be a considerable addition to most Python programmers' toolboxes.

Chapter 2. Branching Using Dictionaries

Functions are objects like everything else in Python, and a function's name is an object reference that refers to the function. If we write a function's name without parentheses, Python knows we mean the object reference, and we can pass such object references around just like any others. We can use this fact to replace `if` statements that have lots of `elif` clauses with a single function call.

In the examples accompanying the book, *Programming in Python 3: A Complete Introduction to the Python Language*,[*] is an interactive console program called `dvds-dbm.py`, that has the following menu:

[*] All the examples are available for download from www.qtrac.eu/py3book.html.

```
(A)dd  (E)dit  (L)ist  (R)emove  (I)mport e(X)port  (Q)uit
```

The program has a function that gets the user's choice and which will return only a valid choice, in this case one of "a", "e", "l", "r", "i", "x", and "q". Here are two equivalent code snippets for calling the relevant function based on the user's choice:

```
if action == "a":
    add_dvd(db)
elif action == "e":
    edit_dvd(db)
elif action == "l":
    list_dvds(db)
elif action == "r":
    remove_dvd(db)
elif action == "i":
    import_(db)
elif action == "x":
    export(db)
elif action == "q":
    quit(db)
```

```
functions = dict(a=add_dvd, e=edit_dvd,
                 l=list_dvds, r=remove_dvd,
                 i=import_, x=export, q=quit)
functions[action](db)
```

The choice is held as a one-character string in the `action` variable, and the database to be used is held in the `db` variable. The `import_()` function has a trailing underscore to keep it distinct from the built-in `import` statement.

In the right-hand code snippet we create a dictionary whose keys are the valid menu choices, and whose values are function references. In the second statement we retrieve the function reference corresponding to the given action and call the function referred to using the call operator, `()`, and in this example, passing the `db` argument. Not only is the code on the right-hand side much shorter than the code on the left, but also it can scale (have far more dictionary items) without affecting its performance, unlike the left-hand code whose speed depends on how many `elif`s must be tested to find the appropriate function to call.

The `convert-incidents.py` program from the book's examples uses this technique in its `import_()` method, as this extract from the method shows:

```
call = {(".aix", "dom"): self.import_xml_dom,
        (".aix", "etree"): self.import_xml_etree,
        (".aix", "sax"): self.import_xml_sax,
        (".ait", "manual"): self.import_text_manual,
        (".ait", "regex"): self.import_text_regex,
        (".aib", None): self.import_binary,
        (".aip", None): self.import_pickle}
result = call[extension, reader](filename)
```

The complete method is 13 lines long; the extension parameter is computed in the method, and the reader is passed in. The dictionary keys are 2-tuples, and the values are methods. If we had used `if` statements, the code would be 22 lines long, and would not scale as well.

Chapter 3. Generator Expressions and Functions

A *generator function* or *generator method* is one which contains a `yield` expression. When a generator function is called it returns an iterator. Values are extracted from the iterator one at a time by calling its `__next__()` method. At each call to `__next__()` the generator function's `yield` expression's value (`None` if none is specified) is returned. If the generator function finishes or executes a `return` a `StopIteration` exception is raised.

In practice we rarely call `__next__()` or catch a `StopIteration`. Instead, we just use a generator like any other iterable. Here are two almost equivalent functions. The one on the left returns a list and the one on the right returns a generator.

```
# Build and return a list
def letter_range(a, z):
    result = []
    while ord(a) < ord(z):
        result.append(a)
        a = chr(ord(a) + 1)
    return result
```

```
# Return each value on demand
def letter_range(a, z):
    while ord(a) < ord(z):
        yield a
        a = chr(ord(a) + 1)
```

We can iterate over the result produced by either function using a `for` loop, for example, `for letter in letter_range("m", "v"):`. But if we want a list of the resultant letters, although calling `letter_range("m", "v")` is sufficient for the left-hand function, for the right-hand generator function we must use `list(letter_range("m", "v"))`.

In addition to generator functions and methods it is also possible to create generator expressions. These are syntactically almost identical to list comprehensions, the difference being that they are enclosed in parentheses rather than brackets. Here are their syntaxes:

(*expression* for *item* in *iterable*)
(*expression* for *item* in *iterable* if *condition*)

Here are two equivalent code snippets that show how a simple `for ... in` loop containing a `yield` expression can be coded as a generator:

```
def items_in_key_order(d):
    for key in sorted(d):
        yield key, d[key]
```

```
def items_in_key_order(d):
    return ((key, d[key])
            for key in sorted(d))
```

Both functions return a generator that produces a list of key–value items for the given dictionary. If we need all the items in one go we can pass the generator returned by the functions to `list()` or `tuple()`; otherwise, we can iterate over the generator to retrieve items as we need them.

Generators provide a means of performing lazy evaluation, which means that they compute only the values that are actually needed. This can be more efficient than, say, computing a very large list in one go. Some generators produce as many values as we ask for—without any upper limit. For example:

```
def quarters(next_quarter=0.0):
    while True:
        yield next_quarter
        next_quarter += 0.25
```

This function will return 0.0, 0.25, 0.5, and so on, forever. Here is how we could use the generator:

```
result = []
for x in quarters():
    result.append(x)
    if x >= 1.0:
        break
```

The `break` statement is essential—without it the `for ... in` loop will never finish. At the end the result list is `[0.0, 0.25, 0.5, 0.75, 1.0]`.

Every time we call `quarters()` we get back a generator that starts at 0.0 and increments by 0.25; but what if we want to reset the generator's current value? It is possible to pass a value into a generator, as this new version of the generator function shows:

```
def quarters(next_quarter=0.0):
    while True:
        received = (yield next_quarter)
        if received is None:
            next_quarter += 0.25
        else:
            next_quarter = received
```

The `yield` expression returns each value to the caller in turn. In addition, if the caller calls the generator's `send()` method, the value sent is received in the generator function as the result of the `yield` expression. Here is how we can use the new generator function:

```
result = []
generator = quarters()
while len(result) < 5:
    x = next(generator)
    if abs(x - 0.5) < sys.float_info.epsilon:
        x = generator.send(1.0)
    result.append(x)
```

We create a variable to refer to the generator and call the built-in `next()` function which retrieves the next item from the generator it is given. (The same effect can be achieved by calling the generator's `__next__()` special method, in this case, `x = generator.__next__()`.) If the value is equal to 0.5 we send the value 1.0 into the generator (which immediately yields this value back). This time the result list is `[0.0, 0.25, 1.0, 1.25, 1.5]`.

In the next subsection we will review the `magic-numbers.py` program which processes files given on the command line. Unfortunately, the Windows shell program (`cmd.exe`) does not provide wildcard expansion (also called *file globbing*), so if a program is run on Windows with the argument `*.*`, the literal text "*.*" will go into the `sys.argv` list instead of all the files in the current directory. We solve this problem by creating two different `get_files()` functions, one for Windows and the other for Unix, both of which use generators. Here's the code:

```
if sys.platform.startswith("win"):
    def get_files(names):
        for name in names:
            if os.path.isfile(name):
                yield name
        else:
            for file in glob.iglob(name):
                if not os.path.isfile(file):
                    continue
                yield file
else:
    def get_files(names):
        return (file for file in names if os.path.isfile(file))
```

In either case the function is expected to be called with a list of filenames, for example, `sys.argv[1:]`, as its argument.

On Windows the function iterates over all the names listed. For each filename, the function yields the name, but for nonfiles (usually directories), the `glob` module's `glob.iglob()` function is used to return an iterator to the names of the files that the name represents after wildcard expansion. For an ordinary name like `autoexec.bat` an iterator that produces one item (the name) is returned, and for a name that uses wildcards like `*.txt` an iterator that produces all the matching files (in this case those with extension `.txt`) is returned. (There is also a `glob.glob()` function that returns a list rather than an iterator.)

On Unix the shell does wildcard expansion for us, so we just need to return a generator for all the files whose names we have been given.[*]

[*] The `glob.glob()` functions are not as powerful as, say, the Unix bash shell, since although they support the `*`, `?`, and `[]` syntaxes, they don't support the `{}` syntax.

Generator functions can be used to create *coroutines*—functions that have multiple entry and exit points (the `yield` expressions) and that can be suspended and resumed at certain points (again at `yield` expressions). Coroutines are often used to provide simpler and lower-overhead alternatives to threading. Several coroutine modules are available from the Python Package Index, `pypi.python.org/pypi`.

Chapter 4. Dynamic Code Execution

There are some occasions when it is easier to write a piece of code that generates the code we need than to write the needed code directly. And in some contexts it is useful to let users enter code (e.g., functions in a spreadsheet), and to let Python execute the entered code for us rather than to write a parser and handle it ourselves—although executing arbitrary code like this is a potential security risk, of course. Another use case for dynamic code execution is to provide plug-ins to extend a program's functionality. Using plug-ins has the disadvantage that all the necessary functionality is not built into the program (which can make the program more difficult to deploy and runs the risk of plug-ins getting lost), but has the advantages that plug-ins can be upgraded individually and can be provided separately, perhaps to provide enhancements that were not originally envisaged.

4.1. Dynamic Code Execution

The easiest way to execute an expression is to use the built-in `eval()` function. For example:

```
x = eval("(2 ** 31) - 1")   # x == 2147483647
```

This is fine for user-entered expressions, but what if we need to create a function dynamically? For that we can use the built-in `exec()` function. For example, the user might give us a formula such as $4\pi r^2$ and the name "area of sphere", which they want turned into a function. Assuming that we replace π with `math.pi`, the function they want can be created like this:

```
import math
code = '''
def area_of_sphere(r):
   return 4 * math.pi * r ** 2
'''
context = {}
context["math"] = math
exec(code, context)
```

We must use proper indentation—after all, the quoted code is standard Python. (Although in this case we could have written it all on a single line because the suite is just one line.)

If `exec()` is called with some code as its only argument there is no way to access any functions or variables that are created as a result of the code being executed. Furthermore, `exec()` cannot access any imported modules or any of the variables, functions, or other objects that are in scope at the point of the call. Both of these problems can be solved by passing a dictionary as the second argument. The dictionary provides a place where object references can be kept for accessing after the `exec()` call has finished. For example, the use of the `context` dictionary means that after the `exec()` call, the dictionary has an object reference to the `area_of_sphere()` function that was created by `exec()`. In this example we needed `exec()` to be able to access the `math` module, so we inserted an item into the context dictionary whose key is the module's name and whose value is an object reference to the corresponding module object. This ensures that inside the `exec()` call, `math.pi` is accessible.

In some cases it is convenient to provide the entire global context to `exec()`. This can be done by passing the dictionary returned by the `globals()` function. One disadvantage of this approach is that any objects created in the `exec()` call would be added to the global dictionary. A solution is to copy the global context into a dictionary, for example, `context = globals().copy()`. This still gives `exec()` access to imported modules and the variables and other objects that are in scope, and because we have copied, any changes to the context made inside the `exec()` call are kept in the `context` dictionary and are not propagated to the global environment. (It would appear to be more secure to use `copy.deepcopy()`, but if security is a concern it is best to avoid `exec()` altogether.) We can also pass the local context, for example, by passing `locals()` as a third argument—this makes objects in the local scope accessible to the code executed by `exec()`.

After the `exec()` call the `context` dictionary contains a key called `"area_of_ sphere"` whose value is the `area_of_sphere()` function. Here is how we can access and call the function:

```
area_of_sphere = context["area_of_sphere"]
area = area_of_sphere(5)          # area == 314.15926535897933
```

The `area_of_sphere` object is an object reference to the function we have dynamically created and can be used just like any other function. And although we created only a single function in the `exec()` call, unlike `eval()`, which can operate on only a single expression, `exec()` can handle as many Python statements as we like, including entire modules, as we will see in the next subsubsection.

4.2. Dynamically Importing Modules

Python provides three straightforward mechanisms that can be used to create plug-ins, all of which involve importing modules by name at runtime. And once we have dynamically imported additional modules, we can use Python's introspection functions to check the availability of the functionality we want, and to access it as required.

In this subsubsection we will review the `magic-numbers.py` program. This program reads the first 1000 bytes of each file given on the command line and for each one outputs the file's type (or the text "Unknown"), and the filename. Here is an example command line and an extract from its output:

```
C:\Python30\python.exe magic-numbers.py c:\windows\*.*
...
XML.................c:\windows\WindowsShell.Manifest
Unknown.............c:\windows\WindowsUpdate.log
Windows Executable..c:\windows\winhelp.exe
Windows Executable..c:\windows\winhlp32.exe
Windows BMP Image...c:\windows\winnt.bmp
...
```

The program tries to load in any module that is in the same directory as the program and whose name contains the text "magic".Such modules are expected to provide a single public function, `get_file_type()`. Two very simple example modules, `StandardMagicNumbers.py` and `WindowsMagicNumbers.py`, that each have a `get_file_type()` function are provided with the book's examples.

We will review the program's `main()` function in two parts.

```
def main():
  modules = load_modules()
  get_file_type_functions = []
  for module in modules:
    get_file_type = get_function(module, "get_file_type")
    if get_file_type is not None:
        get_file_type_functions.append(get_file_type)
```

In a moment, we will look at three different implementations of the `load_modules()` function which returns a (possibly empty) list of module objects, and we will look at the `get_function()` function further on. For each module found we try to retrieve a `get_file_type()` function, and add any we get to a list of such functions.

```
for file in get_files(sys.argv[1:]):
    fh = None
    try:
        fh = open(file, "rb")
        magic = fh.read(1000)
        for get_file_type in get_file_type_functions:
            filetype = get_file_type(magic,
                                     os.path.splitext(file)[1])
            if filetype is not None:
                print("{0:.<20}{1}".format(filetype, file))
                break
        else:
            print("{0:.<20}{1}".format("Unknown", file))

        except EnvironmentError as err:
            print(err)
        finally:
            if fh is not None:
                fh.close()
```

This loop iterates over every file listed on the command line and for each one reads its first 1 000 bytes. It then tries each `get_file_type()` function in turn to see whether it can determine the current file's type. If the file type is determined, the details are printed and the inner loop is broken out of, with processing continuing with the next file. If no function can determine the file type—or if no `get_file_type()` functions were found—an “Unknown” line is printed.

We will now review three different (but equivalent) ways of dynamically importing modules, starting with the longest and most difficult approach, since it shows every step explicitly:

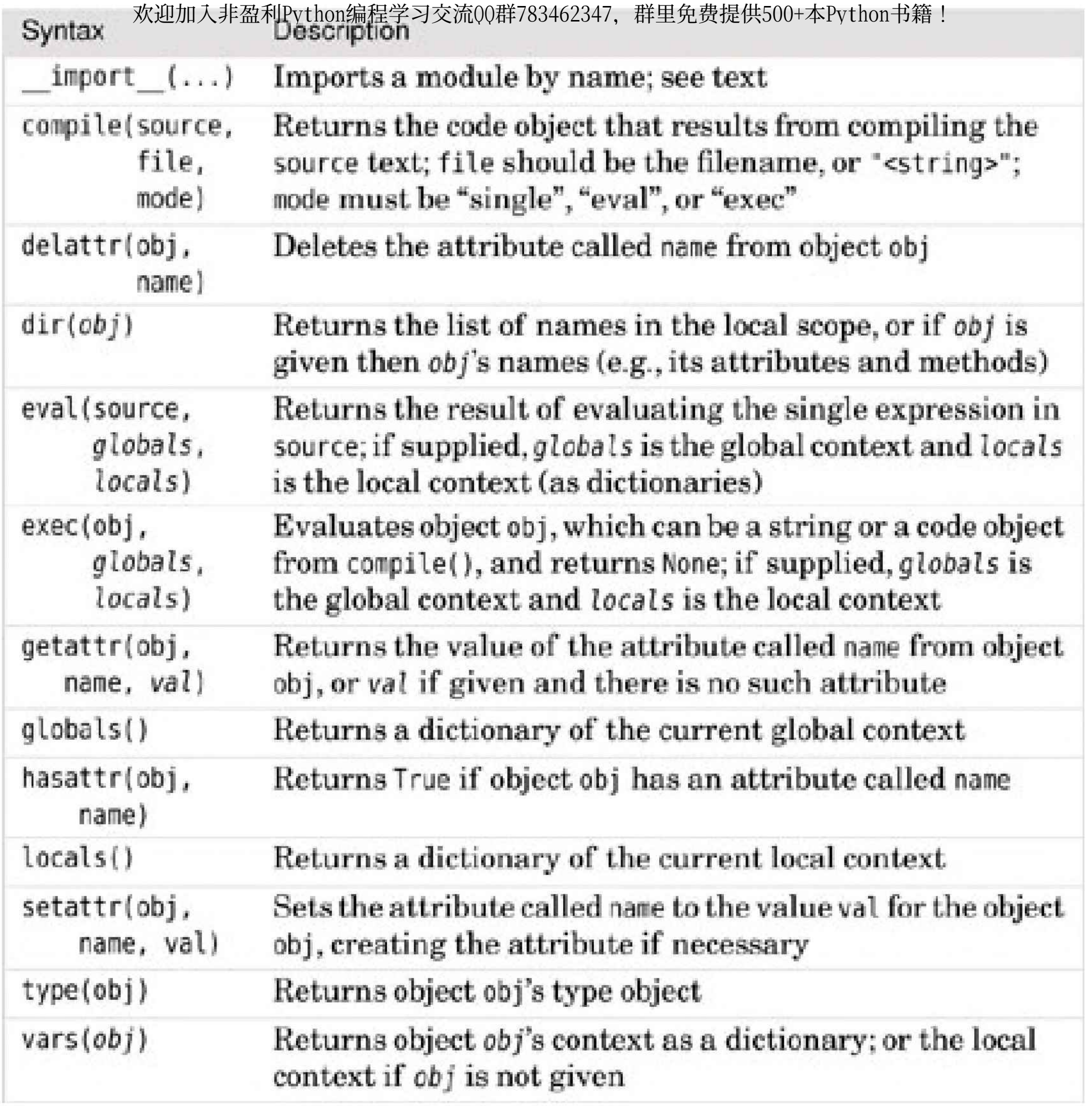
```
def load_modules():
    modules = []
    for name in os.listdir(os.path.dirname(__file__) or "."):
        if name.endswith(".py") and "magic" in name.lower():
            filename = name
            name = os.path.splitext(name)[0]
            if name.isidentifier() and name not in sys.modules:
                fh = None
                try:
                    fh = open(filename, "r", encoding="utf8")
                    code = fh.read()
                    module = type(sys)(name)
                    sys.modules[name] = module
                    exec(code, module.__dict__)
                    modules.append(module)
                except (EnvironmentError, SyntaxError) as err:
                    sys.modules.pop(name, None)
                    print(err)
                finally:
                    if fh is not None:
                        fh.close()
    return modules
```

We begin by iterating over all the files in the program's directory. If this is the current directory, `os.path.dirname(__file__)` will return an empty string which would cause `os.listdir()` to raise an exception, so we pass `"."` if necessary. For each candidate file (ends with `.py` and contains the text "magic"), we get the module name by chopping off the file extension. If the name is a valid identifier it is a viable module name, and if it isn't already in the global list of modules maintained in the `sys.modules` dictionary we can try to import it.

We read the text of the file into the `code` string. The next line, `module = type(sys)(name)`, is quite subtle. When we call `type()` it returns the type object of the object it is given. So if we called `type(1)` we would get `int` back. If we print the type object we just get something human readable like "int", but if we call the type object as a function, we get an object of that type back. For example, we can get the integer 5 in variable `x` by writing `x = 5`, or `x = int(5)`, or `x = type(0)(5)`, or `int_type = type(0); x = int_type(5)`. In this case we've used `type(sys)` and `sys` is a module, so we get back the module type object (essentially the same as a class object), and can use it to create a new module with the given name. Just as with the `int` example where it didn't matter what integer we used to get the `int` type object, it doesn't matter what module we use (as long as it is one that exists, that is, has been imported) to get the module type object.

Once we have a new (empty) module, we add it to the global list of modules to prevent the module from being accidentally reimported. This is done before calling `exec()` to more closely mimic the behavior of the `import` statement. Then we call `exec()` to execute the code we have read—and we use the module's dictionary as the code's context. At the end we add the module to the list of modules we will pass back. And if a problem arises, we delete the module from the global modules dictionary if it has been added—it will not have been added to the list of modules if an error occurred. Notice that `exec()` can handle any amount of code (whereas `eval()` evaluates a single expression—see Table 1), and raises a `SyntaxError` exception if there's a syntax error.

Table 1. Dynamic Programming and Introspection Functions



Syntax	Description
`__import__(...)`	Imports a module by name; see text
`compile(source, file, mode)`	Returns the code object that results from compiling the source text; `file` should be the filename, or `"<string>"`; `mode` must be "single", "eval", or "exec"
`delattr(obj, name)`	Deletes the attribute called `name` from object `obj`
`dir(obj)`	Returns the list of names in the local scope, or if *`obj`* is given then *`obj`*'s names (e.g., its attributes and methods)
`eval(source, globals, locals)`	Returns the result of evaluating the single expression in `source`; if supplied, *`globals`* is the global context and *`locals`* is the local context (as dictionaries)
`exec(obj, globals, locals)`	Evaluates object `obj`, which can be a string or a code object from `compile()`, and returns `None`; if supplied, *`globals`* is the global context and *`locals`* is the local context
`getattr(obj, name, val)`	Returns the value of the attribute called `name` from object `obj`, or *`val`* if given and there is no such attribute
`globals()`	Returns a dictionary of the current global context
`hasattr(obj, name)`	Returns `True` if object `obj` has an attribute called `name`
`locals()`	Returns a dictionary of the current local context
`setattr(obj, name, val)`	Sets the attribute called `name` to the value `val` for the object `obj`, creating the attribute if necessary
`type(obj)`	Returns object `obj`'s type object
`vars(obj)`	Returns object *`obj`*'s context as a dictionary; or the local context if *`obj`* is not given

Here's the second way to dynamically load a module at runtime—the code shown here replaces the first approach's `try` ... `except` block:

```
try:
   exec("import " + name)
   modules.append(sys.modules[name])
except SyntaxError as err:
   print(err)
```

One theoretical problem with this approach is that it is potentially insecure. The `name` variable could begin with `sys;` and be followed by some destructive code.

And here is the third approach, again just showing the replacement for the first approach's `try` ... `except` block:

```
try:
   module = __import__(name)
   modules.append(module)
except (ImportError, SyntaxError) as err:
   print(err)
```

This is the easiest way to dynamically import modules and is slightly safer than using `exec()`, although like any dynamic import, it is by no means secure because we don't know what is being executed when the module is imported.

None of the techniques shown here handles packages or modules in different paths, but it is not difficult to extend the code to accommodate these—although it is worth reading the online documentation, especially for `__import__()`, if more sophistication is required.

Having imported the module we need to be able to access the functionality it provides. This can be achieved using Python's built-in introspection functions, `getattr()` and `hasattr()`. Here's how we have used them to implement the `get_function()` function:

```
def get_function(module, function_name):
   function = get_function.cache.get((module, function_name), None)
   if function is None:
      try:
         function = getattr(module, function_name)
         if not hasattr(function, "__call__"):
            raise AttributeError()
         get_function.cache[module, function_name] = function
      except AttributeError:
         function = None
   return function
get_function.cache = {}
```

Ignoring the cache-related code for a moment, what the function does is call `getattr()` on the module object with the name of the function we want. If there is no such attribute an `AttributeError` exception is raised, but if there is such an attribute we use `hasattr()` to check that the attribute itself has the `__call__` attribute—something that all callables (functions and methods) have. (Further on we will see a nicer way of checking whether an attribute is callable.) If the attribute exists and is callable we can return it to the caller; otherwise, we return `None` to signify that the function isn't available.

collections. Callable

☞55

If hundreds of files were being processed (e.g., due to using *.* in the `C:\windows` directory), we don't want to go through the lookup process for every module for every file. So immediately after defining the `get_function()` function, we add an attribute to the function, a dictionary called `cache`. (In general, Python allows us to add arbitrary attributes to arbitrary objects.) The first time that `get_function()` is called the cache dictionary is empty, so the `dict.get()` call will return `None`. But each time a suitable function is found it is put in the dictionary with a 2-tuple of the module and function name used as the key and the function itself as the value. So the second and all subsequent times a particular function is requested the function is immediately returned from the cache and no attribute lookup takes place at all.[*]

[*] A slightly more sophisticated `get_function()` that has better handling of modules without the required functionality is in the `magic-numbers.py` program alongside the version shown here.

The technique used for caching the `get_function()`'s return value for a given set of arguments is called *memoizing*. It can be used for any function that has no side effects (does not change any global variables), and that always returns the same result for the same (immutable) arguments. Since the code required to create and manage a cache for each memoized function is the same, it is an ideal candidate for a function decorator, and several `@memoize` decorator recipes are given in the Python Cookbook, in `code.activestate.com/recipes/langs/python/`. However, module objects are mutable, so some off-the-shelf memoizer decorators wouldn't work with our `get_function()` function as it stands. An easy solution would be to use each module's `__name__` string rather than the module itself as the first part of the key tuple.

Doing dynamic module imports is easy, and so is executing arbitrary Python code using the `exec()` function. This can be very convenient, for example, allowing us to store code in a database. However, we have no control over what imported or `exec()`uted code will do. Recall that in addition to variables, functions, and classes, modules can also contain code that is executed when it is imported—if the code came from an untrusted source it might do something unpleasant. How to address this depends on circumstances, although it may not be an issue at all in some environments, or for personal projects.

Chapter 5. Local and Recursive Functions

It is often useful to have one or more small helper functions inside another function. Python allows this without formality—we simply define the functions we need inside the definition of an existing function. Such functions are often called *nested functions* or *local functions*.

One common use case for local functions is when we want to use recursion. In these cases, the enclosing function is called, sets things up, and then makes the first call to a local recursive function. Recursive functions (or methods) are ones that call themselves. Structurally, all directly recursive functions can be seen as having two cases: the *base case* and the *recursive case*. The base case is used to stop the recursion.

Recursive functions can be computationally expensive because for every recursive call another stack frame is used; however, some algorithms are most naturally expressed using recursion. Most Python implementations have a fixed limit to how many recursive calls can be made. The limit is returned by `sys.getrecursionlimit()` and can be changed by `sys.setrecursionlimit()`, although increasing the limit is most often a sign that the algorithm being used is inappropriate or that the implementation has a bug.

The classic example of a recursive function is one that is used to calculate factorials.[*] For example, `factorial(5)` will calculate 5! and return 120, that is, 1 × 2 × 3 × 4 × 5:

[*] Python's `math` module provides a much more efficient `math.factorial()` function.

```
def factorial(x):
   if x <= 1:
      return 1
   return x * factorial(x - 1)
```

This is not an efficient solution, but it does show the two fundamental features of recursive functions. If the given number, `x`, is 1 or less, 1 is returned and no recursion occurs—this is the base case. But if `x` is greater than 1 the value returned is `x * factorial(x - 1)`, and this is the recursive case because here the factorial function calls itself. The function is guaranteed to terminate because if the initial `x` is less than or equal to 1 the base case will be used and the function will finish immediately, and if `x` is greater than 1, each recursive call will be on a number one less than before and so will eventually be 1.

To see both local functions and recursive functions in a meaningful context we will study the `indented_list_sort()` function from module file `IndentedList.py`. This function takes a list of strings that use indentation to create a hierarchy, and a string that holds one level of indent, and returns a list with the same strings but where all the strings are sorted in case-insensitive alphabetical order, with indented items sorted under their parent item, recursively, as the `before` and `after` lists shown in Figure 1 illustrate.

Figure 1. Before and after sorting an indented list

```
before = ["Nonmetals",
          "    Hydrogen",
          "    Carbon",
          "    Nitrogen",
          "    Oxygen",
          "Inner Transitionals",
          "    Lanthanides",
          "        Cerium",
          "        Europium",
          "    Actinides",
          "        Uranium",
          "        Curium",
          "        Plutonium",
          "Alkali Metals",
          "    Lithium",
          "    Sodium",
          "    Potassium"]
```

```
after = ["Alkali Metals",
         "    Lithium",
         "    Potassium",
         "    Sodium",
         "Inner Transitionals",
         "    Actinides",
         "        Curium",
         "        Plutonium",
         "        Uranium",
         "    Lanthanides",
         "        Cerium",
         "        Europium",
         "Nonmetals",
         "    Carbon",
         "    Hydrogen",
         "    Nitrogen",
         "    Oxygen"]
```

Given the `before` list, the `after` list is produced by this call: `after = IndentedList.indented_list_sort(before)`. The default indent value is four spaces, the same as the indent used in the `before` list, so we did not need to set it explicitly.

We will begin by looking at the `indented_list_sort()` function as a whole, and then we will look at its two local functions.

```
def indented_list_sort(indented_list, indent="    "):
    KEY, ITEM, CHILDREN = range(3)

    def add_entry(level, key, item, children):
        ...

    def update_indented_list(entry):
        ...

    entries = []
    for item in indented_list:
        level = 0
        i = 0
        while item.startswith(indent, i):
            i += len(indent)
            level += 1
        key = item.strip().lower()
        add_entry(level, key, item, entries)

    indented_list = []
    for entry in sorted(entries):
        update_indented_list(entry)
    return indented_list
```

The code begins by creating three constants that are used to provide names for index positions used by the local functions. Then we define the two local functions which we will review in a moment. The sorting algorithm works in two stages. In the first stage we create a list of entries, each a 3-tuple consisting of a “key” that will be used for sorting, the original string, and a list of the string’s child entries. The key is just a lowercased copy of the string with whitespace stripped from both ends. The level is the indentation level, 0 for top-level items, 1 for children of top-level items, and so on. In the second stage we create a new indented list and add each string from the sorted entries list, and each string’s child strings, and so on, to produce a sorted indented list.

```
def add_entry(level, key, item, children):
    if level == 0:
        children.append((key, item, []))
    else:
        add_entry(level - 1, key, item, children[-1][CHILDREN])
```

This function is called for each string in the list. The `children` argument is the list to which new entries must be added. When called from the outer function (`indented_list_sort()`), this is the `entries` list. This has the effect of turning a list of strings into a list of entries, each of which has a top-level (unindented) string and a (possibly empty) list of child entries.

If the level is 0 (top-level), we add a new 3-tuple to the `entries` list. This holds the key (for sorting), the original item (which will go into the resultant sorted list), and an empty children list. This is the base case since no recursion takes place. If the level is greater than 0, the item is a child (or descendant) of the last item in the `children` list. In this case we recursively call `add_entry()` again, reducing the level by 1 and passing the children list's last item's children list as the list to add to. If the level is 2 or more, more recursive calls will take place, until eventually the level is 0 and the children list is the right one for the entry to be added to.

For example, when the "Inner Transitionals" string is reached, the outer function calls `add_entry()` with a level of 0, a key of "inner transitionals", an item of "Inner Transitionals", and the `entries` list as the children list. Since the level is 0, a new item will be appended to the children list (`entries`), with the key, item, and an empty children list. The next string is "Lanthanides"—this is indented, so it is a child of the "Inner Transitionals" string. The `add_entry()` call this time has a level of 1, a key of "lanthanides", an item of "Lanthanides", and the `entries` list as the children list. Since the level is 1, the `add_entry()` function calls itself recursively, this time with level 0 (1 - 1), the same key and item, but with the children list being the children list of the last item, that is, the "Inner Transitionals" item's children list.

Here is what the `entries` list looks like once all the strings have been added, but before the sorting has been done:

```
[('nonmetals',
  'Nonmetals',
  [('hydrogen', '    Hydrogen', []),
   ('carbon', '    Carbon', []),
   ('nitrogen', '    Nitrogen', []),
  ('oxygen', '    Oxygen', [])]),
 ('inner transitionals',
  'Inner Transitionals',
  [('lanthanides',
    '    Lanthanides',
    [('cerium', '        Cerium', []),
     ('europium', '        Europium', [])]),
   ('actinides',
    '    Actinides',
    [('uranium', '        Uranium', []),
     ('curium', '        Curium', []),
     ('plutonium', '        Plutonium', [])])]),
 ('alkali metals',
  'Alkali Metals',
  [('lithium', '    Lithium', []),
   ('sodium', '    Sodium', []),
   ('potassium', '    Potassium', [])])]
```

The output was produced using the `pprint` ("pretty print") module's `pprint.pprint()` function. Notice that the `entries` list has only three items (all of which are 3-tuples), and that each 3-tuple's last element is a list of child 3-tuples (or is an empty list).

The `add_entry()` function is both a local function and a recursive function. Like all recursive functions, it has a *base case* (in this function, when the level is 0) that ends the recursion, and a recursive case.

The function could be written in a slightly different way:

```
def add_entry(key, item, children):
    nonlocal level
    if level == 0:
        children.append((key, item, []))
    else:
        level -= 1
        add_entry(key, item, children[-1][CHILDREN])
```

Here, instead of passing `level` as a parameter, we use a `nonlocal` statement to access a variable in an outer enclosing scope. If we did not change `level` inside the function we would not need the `nonlocal` statement—in such a situation, Python would not find it in the local (inner function) scope, and would look at the enclosing scope and find it there. But in this version of `add_entry()` we need to change level's value, and just as we need to tell Python that we want to change global variables using the global statement (to prevent a new local variable from being created rather than the global variable updated), the same applies to variables that we want to change but which belong to an outer scope. Although it is often best to avoid using global altogether, it is also best to use nonlocal with care.

```
def update_indented_list(entry):
    indented_list.append(entry[ITEM])
    for subentry in sorted(entry[CHILDREN]):
        update_indented_list(subentry)
```

In the algorithm's first stage we build up a list of entries, each a (key, item, children) 3-tuple, in the same order as they are in the original list. In the algorithm's second stage we begin with a new empty indented list and iterate over the sorted entries, calling update_indented_list() for each one to build up the new indented list. The update_indented_list() function is recursive. For each top-level entry it adds an item to the indented_list, and then calls itself for each of the item's child entries. Each child is added to the indented_list, and then the function calls itself for each child's children—and so on. The base case (when the recursion stops) is when an item, or child, or child of a child, and so on has no children of its own.

Python looks for indented_list in the local (inner function) scope and doesn't find it, so it then looks in the enclosing scope and finds it there. But notice that inside the function we append items to the indented_list even though we have not used nonlocal. This works because nonlocal (and global) are concerned with object references, not with the objects they refer to. In the second version of add_entry() we had to use nonlocal for level because the += operator applied to a number rebinds the object reference to a new object—what really happens is level = level + 1, so level is set to refer to a new integer object. But when we call list.append() on the indented_list, it modifies the list itself and no rebinding takes place, and therefore nonlocal is not necessary. (For the same reason, if we have a dictionary, list, or other global collection, we can add or remove items from it without using a global statement.)

Chapter 6. Function and Method Decorators

A decorator is a function that takes a function or method as its sole argument and returns a new function or method that incorporates the decorated function or method with some additional functionality added. We have already made use of some predefined decorators, for example, `@property` and `@classmethod`. In this section we will learn how to create our own function decorators, and later in this short cut we will see how to create class decorators.

Class decorators

☞42

For our first decorator example, let us suppose that we have many functions that perform calculations, and that some of these must always produce a positive result. We could add an assertion to each of these, but using a decorator is easier and clearer. Here's a function decorated with the `@positive_result` decorator that we will create in a moment:

```
@positive_result
def discriminant(a, b, c):
    return (b ** 2) - (4 *a*c)
```

Thanks to the decorator, if the result is ever less than 0, an `AssertionError` exception will be raised and the program will terminate. And of course, we can use the decorator on as many functions as we like. Here's the decorator's implementation:

```
def positive_result(function):
    def wrapper(*args, **kwargs):
        result = function(*args, **kwargs)
        assert result >= 0, function.__name__ + "() result isn't >= 0"
        return result
    wrapper.__name__ = function.__name__
    wrapper.__doc__ = function.__doc__
    return wrapper
```

Decorators define a new local function that calls the original function. Here, the local function is `wrapper()`; it calls the original function and stores the result, and it uses an assertion to guarantee that the result is positive (or that the program will terminate). The wrapper finishes by returning the result computed by the wrapped function. After creating the wrapper, we set its name and docstring to those of the original function. This helps with introspection, since we want error messages to mention the name of the original function, not the wrapper. Finally, we return the wrapper function—it is this function that will be used in place of the original.

```
def positive_result(function):
   @functools.wraps(function)
   def wrapper(*args, **kwargs):
      result = function(*args, **kwargs)
      assert result >= 0, function.__name__ + "() result isn't >= 0"
      return result
   return wrapper
```

Here is a slightly cleaner version of the `@positive_result` decorator. The wrapper itself is wrapped using the `functools` module's `@functools.wraps` decorator, which ensures that the `wrapper()` function has the name and docstring of the original function.

In some cases it would be useful to be able to parameterize a decorator, but at first sight this does not seem possible since a decorator takes just one argument, a function or method. But there is a neat solution to this. We can call a function with the parameters we want and that returns a decorator which can then decorate the function that follows it. For example:

```
@bounded(0, 100)
def percent(amount, total):
   return (amount / total) * 100
```

Here, the `bounded()` function is called with two arguments, and returns a decorator that is used to decorate the `percent()` function. The purpose of the decorator in this case is to guarantee that the number returned is always in the range 0 to 100 inclusive. Here's the implementation of the `bounded()` function:

```
def bounded(minimum, maximum):
   def decorator(function):
      @functools.wraps(function)
```

```
    def wrapper(*args, **kwargs):
        result = function(*args, **kwargs)
        if result < minimum:
            return minimum
        elif result > maximum:
            return maximum

            return result
        return wrapper
    return decorator
```

The function creates a decorator function, that itself creates a wrapper function. The wrapper performs the calculation and returns a result that is within the bounded range. The `decorator()` function returns the `wrapper()` function, and the `bounded()` function returns the decorator.

One further point to note is that each time a wrapper is created inside the `bounded()` function, the particular wrapper uses the minimum and maximum values that were passed to `bounded()`.

The last decorator we will create in this subsection is a bit more complex. It is a logging function that records the name, arguments, and result of any function it is used to decorate. For example:

```
@logged
def discounted_price(price, percentage, make_integer=False):
    result = price * ((100 - percentage) / 100)
    if not (0 < result <= price):
        raise ValueError("invalid price")
    return result if not make_integer else int(round(result))
```

If Python is run in debug mode (the normal mode), every time the `discount-ed_price()` function is called a log message will be added to the file `logged.log` in the machine's local temporary directory, as this log file extract illustrates:

```
called: discounted_price(100, 10) -> 90.0
called: discounted_price(210, 5) -> 199.5
called: discounted_price(210, 5, make_integer=True) -> 200
called: discounted_price(210, 14, True) -> 181
called: discounted_price(210, -8) <type 'ValueError'>: invalid price
```

If Python is run in optimized mode (using the `-o` command-line option or if the `PYTHONOPTIMIZE` environment variable is set to `-o`), then no logging will take place. Here’s the code for setting up logging and for the decorator:

```
if __debug__:
    logger = logging.getLogger("Logger")
    logger.setLevel(logging.DEBUG)
    handler = logging.FileHandler(os.path.join(
                                tempfile.gettempdir(), "logged.log"))

    logger.addHandler(handler)

    def logged(function):
        @functools.wraps(function)
        def wrapper(*args, **kwargs):

        log = "called: " + function.__name__ + "("
        log += ", ".join(["{0!r}".format(a) for a in args] +
                         ["{0!s}={1!r}".format(k, v)
                          for k, v in kwargs.items()])
        result = exception = None
        try:
            result = function(*args, **kwargs)
            return result
        except Exception as err:
            exception = err
        finally:
            log += ((") -> " + str(result)) if exception is None
                    else ") {0}: {1}".format(type(exception),
                                            exception))
            logger.debug(log)
            if exception is not None:
                raise exception
        return wrapper
else:
    def logged(function):
        return function
```

In debug mode the global variable `__debug__` is `True`. If this is the case we set up logging using the `logging` module, and then create the `@logged` decorator. The logging module is very powerful and flexible—it can log to files, rotated files, emails, network connections, HTTP servers, and more. Here we’ve used only the most basic facilities by creating a logging object, setting its logging level (several levels are supported), and choosing to use a file for the output.

The wrapper's code begins by setting up the log string with the function's name and arguments. We then try calling the function and storing its result. If any exception occurs we store it. In all cases the `finally` block is executed, and there we add the return value (or exception) to the log string and write to the log. If no exception occurred, the result is returned; otherwise, we reraise the exception to correctly mimic the original function's behavior.

If Python is running in optimized mode, `__debug__` is `False`; in this case we define the `logged()` function to simply return the function it is given, so apart from the tiny overhead of this indirection when the function is first created, there is no runtime overhead at all.

Note that the standard library's `trace` and `profile` modules can run and analyse programs and modules to produce various tracing and profiling reports. Both use introspection, so unlike the `@logged` decorator we have used here, neither `trace` nor `profile` requires any source code changes.

Chapter 7. Function Annotations

Functions and methods can be defined with annotations—expressions that can be used in a function's signature. Here's the general syntax:

```
def functionName(par1 : exp1, par2 : exp2, ..., parN : expN)-> rexp:
    suite
```

Every colon expression part (`: expX`) is an optional annotation, and so is the arrow return expression part (`->rexp`). The last (or only) positional parameter (if present) can be of the form `*args`, with or without an annotation; similarly, the last (or only) keyword parameter (if present) can be of the form `**kwargs`, again with or without an annotation.

If annotations are present they are added to the function's `__annotations__` dictionary; if they are not present this dictionary is empty. The dictionary's keys are the parameter names, and the values are the corresponding expressions. The syntax allows us to annotate all, some, or none of the parameters and to annotate the return value or not. Annotations have no special significance to Python. The only thing that Python does in the face of annotations is to put them in the `__annotations__` dictionary; any other action is up to us. Here is an example of an annotated function that is in the `Util` module:

```
def is_unicode_punctuation(s : str) -> bool:
    for c in s:
        if unicodedata.category(c)[0] != "P":
            return False
    return True
```

Every Unicode character belongs to a particular category and each category is identified by a two-character identifier. All the categories that begin with *P* are punctuation characters.

Here we have used Python data types as the annotation expressions. But they have no particular meaning for Python, as these calls should make clear:

```
Util.is_unicode_punctuation("zebr\a")      # returns: False
Util.is_unicode_punctuation(s="!@#?")      # returns: True
Util.is_unicode_punctuation(("!", "@"))    # returns: True
```

The first call uses a positional argument and the second call a keyword argument, just to show that both kinds work as expected. The last call passes a tuple rather than a string, and this is accepted since Python does nothing more than record the annotations in the `__annotations__` dictionary.

If we want to give meaning to annotations, for example, to provide type checking, one approach is to decorate the functions we want the meaning to apply to with a suitable decorator. Here is a very basic type-checking decorator:

```
def strictly_typed(function):
    annotations = function.__annotations__
    arg_spec = inspect.getfullargspec(function)

    assert "return" in annotations, "missing type for return value"
    for arg in arg_spec.args + arg_spec.kwonlyargs:
        assert arg in annotations, ("missing type for parameter '" +
                                    arg + "'")
    @functools.wraps(function)
    def wrapper(*args, **kwargs):
        for name, arg in (list(zip(arg_spec.args, args)) +
                          list(kwargs.items())):
            assert isinstance(arg, annotations[name]), (
                    "expected argument '{0}' of {1} got {2}".format(
                    name, annotations[name], type(arg)))
        result = function(*args, **kwargs)
        assert isinstance(result, annotations["return"]), (
                    "expected return of {0} got {1}".format(
                    annotations["return"], type(result)))
        return result
    return wrapper
```

This decorator requires that every argument and the return value must be annotated with the expected type. It checks that the function’s arguments and return type are all annotated with their types when the function it is passed is created, and at runtime it checks that the types of the actual arguments match those expected.

The `inspect` module provides powerful introspection services for objects. Here, we have made use of only a small part of the argument specification object it returns, to get the names of each positional and keyword argument—in the correct order in the case of the positional arguments. These names are then used in conjunction with the annotations dictionary to ensure that every parameter and the return value are annotated.

The wrapper function created inside the decorator begins by iterating over every name–argument pair of the given positional and keyword arguments. Since `zip()` returns an iterator and `dictionary.items()` returns a dictionary view we cannot concatenate them directly, so first we convert them both to lists. If any actual argument has a different type from its corresponding annotation the assertion will fail; otherwise, the actual function is called and the type of the value returned is checked, and if it is of the right type, it is returned. At the end of the `strictly_typed()` function, we return the wrapped function as usual.

Notice that the checking is done only in debug mode (which is Python's default mode—controlled by the `-O` command-line option and the `PYTHONOPTIMIZE` environment variable).

If we decorate the `is_unicode_punctuation()` function with the `@strictly_typed` decorator, and try the same examples as before using the decorated version, the annotations are acted upon:

```
is_unicode_punctuation('zebr\a')          # returns: False
is_unicode_punctuation(s="!@#?")          # returns: True
is_unicode_punctuation(("!", "@"))        # raises AssertionError
```

Now the argument types are checked, so in the last case an `AssertionError` is raised because a tuple is not a string or a subclass of `str`.

Now we will look at a completely different use of annotations. Here's a small function that has the same functionality as the built-in `range()` function, except that it always returns `float`s:

```
def range_of_floats(*args) -> "author=Reginald Perrin":
    return (float(x) for x in range(*args))
```

No use is made of the annotation by the function itself, but it is easy to envisage a tool that imported all of a project's modules and produced a list of function names and author names, extracting each function's name from its `__name__` attribute, and the author names from the value of the `__annotations__` dictionary's `"return"` item.

Annotations are a very new feature of Python, and because Python does not impose any predefined meaning on them, the uses they can be put to are limited only by our imagination. Further ideas for possible uses, and some useful links, are available from PEP 3107 "Function Annotations", www.python.org/dev/peps/pep-3107.

Chapter 8. Controlling Attribute Access

Let's start with one very small and simple new feature. Here is the start of the definition of a simple `Point` class:

```
class Point:

    __slots__ = ("x", "y")

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
```

When a class is created without the use of `__slots__`, behind the scenes Python creates a private dictionary called `__dict__` for each instance, and this dictionary holds the instance's data attributes. This is why we can add or remove attributes from objects. (For example, we added a `cache` attribute to the `get_function()` function earlier in this short cut.)

Attribute access functions

14

If we only need objects where we access the original attributes and don't need to add or remove attributes, we can create classes that don't have a `__dict__`. This is achieved simply by defining a class attribute called `__slots__` whose value is a tuple of attribute names. Each object of such a class will have attributes of the specified names and no `__dict__`; no attributes can be added or removed from such classes. These objects consume less memory than conventional objects, although this is unlikely to make much difference unless large numbers of objects are created.

Having seen how to limit a class's attributes, we will now look at an example where attribute values are computed on the fly rather than stored. Here's the complete implementation of such a class:

```
class Ord:
    def __getattr__(self, char):
        return ord(char)
```

With the `Ord` class available, we can create an instance, `ord = Ord()`, and then have an alternative to the built-in `ord()` function that works for any character that is a valid identifier. For example, `ord.a` returns 97, `ord.Z` returns 90, and `ord.å` returns 229. (But `ord.!` and similar are syntax errors.)

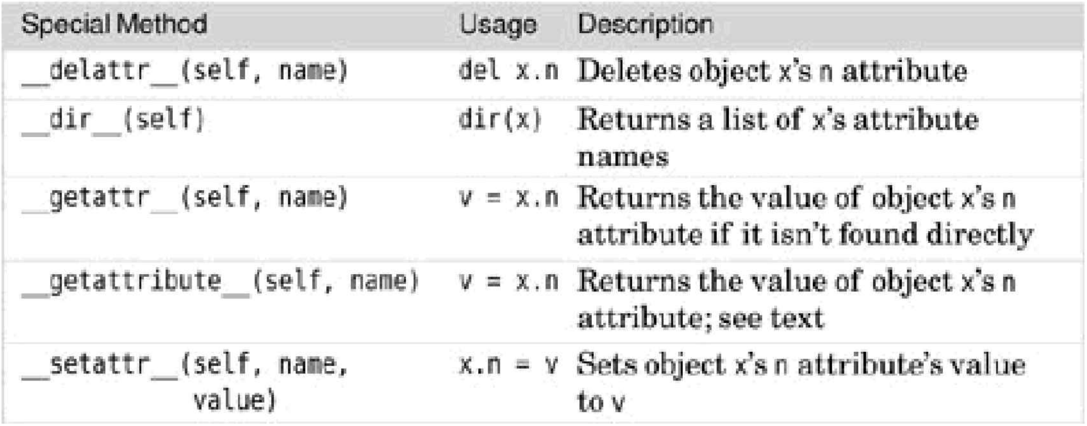
Note that if we typed the `Ord` class into IDLE it would not work if we then typed `ord = Ord()`. This is because the instance has the same name as the built-in `ord()` function that the `Ord` class uses, so the `ord()` call would actually become a call to the `ord` instance and result in a `TypeError` exception. The problem would not arise if we imported a module containing the `Ord` class because the interactively created `ord` object and the built-in `ord()` function used by the `Ord` class would be in two separate modules, so one would not displace the other. If we really need to create a class interactively and to reuse the name of a built-in we can do so by ensuring that the class calls the built-in—in this case by importing the `builtins` module which provides unambiguous access to all the built-in functions, and calling `builtins.ord()` rather than plain `ord()`.

Here's another tiny yet complete class. This one allows us to create "constants". It isn't difficult to change the values behind the class's back, but it can at least prevent simple mistakes.

```
class Const:

    def __setattr__(self, name, value):
        if name in self.__dict__:

            raise ValueError("cannot change a const attribute")
        self.__dict__[name] = value

    def __delattr__(self, name):
        if name in self.__dict__:
            raise ValueError("cannot delete a const attribute")
        raise AttributeError("'{0}' object has no attribute '{1}'"
                             .format(self.__class__.__name__, name))
```

With this class we can create a constant object, say, `const = Const()`, and set any attributes we like on it, for example, `const.limit = 591`. But once an attribute's value has been set, although it can be read as often as we like, any attempt to change or delete it will result in a `ValueError` exception being raised. We have not reimplemented `__getattr__()` because the base class `object.__getattr__()` method does what we want—returns the given attribute's value or raises an `AttributeError` exception if there is no such attribute. In the `__delattr__()` method we mimic the `__getattr__()` method's error message for nonexistent attributes, and to do this we must get the name of the class we are in as well as the name of the nonexistent attribute. The class works because we are using the object's `__dict__` which is what the base class `__getattr__()`, `__setattr__()`, and `__delattr__()` methods use, although here we have used only the base class's `__getattr__()` method. All the special methods used for attribute access are listed in Table 2.

Table 2. Attribute Access Special Methods



Special Method	Usage	Description
`__delattr__(self, name)`	`del x.n`	Deletes object x's n attribute
`__dir__(self)`	`dir(x)`	Returns a list of x's attribute names
`__getattr__(self, name)`	`v = x.n`	Returns the value of object x's n attribute if it isn't found directly
`__getattribute__(self, name)`	`v = x.n`	Returns the value of object x's n attribute; see text
`__setattr__(self, name, value)`	`x.n = v`	Sets object x's n attribute's value to v

There is another way of getting constants: We can use named tuples. Here are a couple of examples:

```
Const = collections.namedtuple("_", "min max")(191, 591)
Const.min, Const.max                    # returns: (191, 591)
Offset = collections.namedtuple("_", "id name description")(*range(3))
Offset.id, Offset.name, Offset.description  # returns: (0, 1, 2)
```

In both cases we have just used a throwaway name for the named tuple because we want just one named tuple instance each time, not a tuple subclass for creating instances of a named tuple. Although Python does not support an enum data type, we can use named tuples as we have done here to get a similar effect.

For our last look at attribute access special methods we will use the book's `Image.py` example. This module defines an `Image` class whose width, height, and background color are fixed when an `Image` is created (although they are changed if an image is loaded). We provided access to them using read-only properties. For example, we had:

```
@property
def width(self):
    return self.__width
```

This is easy to code but could become tedious if there are a lot of read-only properties. Here is a different solution that handles all the `Image` class's read-only properties in a single method:

```
def __getattr__(self, name):
    if name == "colors":
        return set(self.__colors)
    classname = self.__class__.__name__
    if name in frozenset({"background", "width", "height"}):
        return self.__dict__["_{0}__{1}".format(classname, name)]
    raise AttributeError("'{0}' object has no attribute '{1}'"
                         .format(classname, name))
```

If we attempt to access an object's attribute and the attribute is not found, Python will call the `__getattr__()` method (providing it is implemented, and that we have not reimplemented `__getattribute__()`), with the name of the attribute as a parameter. Implementations of `__getattr__()` must raise an `AttributeError` exception if they do not handle the given attribute.

For example, if we have the statement `image.colors`, Python will look for a `colors` attribute and having failed to find it, will then call `Image.__getattr__(image, "colors")`. In this case the `__getattr__()` method handles a `"colors"` attribute name and returns a copy of the set of colors that the image is using.

The other attributes are immutable, so they are safe to return directly to the caller. We could have written separate `elif` statements for each one like this:

```
elif name == "background":
    return self.__background
```

But instead we have chosen a more compact approach. Since we know that under the hood all of an object's nonspecial attributes are held in `self.__dict__`, we have chosen to access them directly. For private attributes (those whose name begins with two leading underscores), the name is mangled to have the form *`_className__attributeName`*, so we must account for this when retrieving the attribute's value from the object's private dictionary.

For the name mangling needed to look up private attributes and to provide the standard `AttributeError` error text, we need to know the name of the class we are in. (It may not be `Image` because the object might be an instance of an `Image` subclass.) Every object has a `__class__` special attribute, so `self.__class__` is always available inside methods and can safely be accessed by `__getattr__()` without risking unwanted recursion.

Note that there is a subtle difference in that using `__getattr__()` and `self.__class__` provides access to the attribute in the instance's class (which may be a subclass), but accessing the attribute directly uses the class the attribute is defined in.

One special method that we have not covered is `__getattribute__()`. Whereas the `__getattr__()` method is called last when looking for (nonspecial) attributes, the `__getattribute__()` method is called first for every attribute access. Although it can be useful or even essential in some cases to call `__getattribute__()`, reimplementing the `__getattribute__()` method can be tricky. Reimplementations must be very careful not to call themselves recursively—using `super().__getattribute__()` or `object.__getattribute__()` is often done in such cases. Also, since `__getattribute__()` is called for every attribute access, reimplementing it can easily end up degrading performance compared with direct attribute access or properties. None of the classes presented in this book reimplements `__getattribute__()`.

Chapter 9. Functors

In Python a *function object* is an object reference to any callable, such as a function, a lambda function, or a method. The definition also includes classes, since an object reference to a class is a callable that, when called, returns an object of the given class—for example, `x = int(5)`. In computer science a *functor* is an object that can be called as though it were a function, so in Python terms a functor is just another kind of function object. Any class that has a `__call__()` special method is a functor. The key benefit that functors offer is that they can maintain some state information. For example, we could create a functor that always strips basic punctuation from the ends of a string. We would create and use it like this:

```
strip_punctuation = Strip(",;:.!?")
strip_punctuation("Land ahoy!")      # returns: 'Land ahoy'
```

Here we create an instance of the `Strip` functor initializing it with the value `",;:.!?"`. Whenever the instance is called it returns the string it is passed with any punctuation characters stripped off. Here's the complete implementation of the `Strip` class:

```
class Strip:

  def __init__(self, characters):
    self.characters = characters

  def __call__(self, string):
    return string.strip(self.characters)
```

We could achieve the same thing using a plain function or lambda, but if we need to store a bit more state or perform more complex processing, a functor is often the right solution.

A functor's ability to capture state by using a class is very versatile and powerful, but sometimes it is more than we really need. Another way to capture state is to use a *closure*. A closure is a function or method that captures some external state. For example:

```
def make_strip_function(characters):
    def strip_function(string):
        return string.strip(characters)
    return strip_function

strip_punctuation = make_strip_function(",;:.!?")
strip_punctuation("Land ahoy!")       # returns: 'Land ahoy'
```

The `make_strip_function()` function takes the characters to be stripped as its sole argument and returns a function, `strip_function()`, that takes a string argument and which strips the characters that were given at the time the closure was created. So just as we can create as many instances of the `Strip` class as we want, each with its own characters to strip, we can create as many strip functions with their own characters as we like.

The classic use case for functors is to provide key functions for sort routines. Here is a generic `SortKey` functor class (from file `SortKey.py`):

```
class SortKey:

    def __init__(self, *attribute_names):
        self.attribute_names = attribute_names

    def __call__(self, instance):
        values = []
        for attribute_name in self.attribute_names:

            values.append(getattr(instance, attribute_name))
        return values
```

When a `SortKey` object is created it keeps a tuple of the attribute names it was initialized with. When the object is called it creates a list of the attribute values for the instance it is passed—in the order they were specified when the `SortKey` was initialized. For example, imagine we have a `Person` class:

```
class Person:

    def __init__(self, forename, surname, email):
        self.forename = forename
        self.surname = surname
        self.email = email
```

Suppose we have a list of `Person` objects in the `people` list. We can sort the list by surnames like this: `people.sort(key=SortKey("surname"))`. If there are a lot of people there are bound to be some surname clashes, so we can sort by surname, and then by forename within surname, like this: `people.sort(key=SortKey("surname", "forename"))`. And if we had people with the same surname and forename we could add the email attribute too. And of course, we could sort by forename and then surname by changing the order of the attribute names we give to the `SortKey` functor.

Another way of achieving the same thing, but without needing to create a functor at all, is to use the `operator` module's `operator.attrgetter()` function. For example, to sort by surname we could write: `people.sort(key=operator.attrgetter("surname"))`. And similarly, to sort by surname and forename: `people.sort(key=operator.attrgetter("surname", "forename"))`. The `operator. attrgetter()` function returns a function (a closure) that, when called on an object, returns those attributes of the object that were specified when the closure was created.

Functors are probably used rather less frequently in Python than in other languages that support them because Python has other means of doing the same things—for example, using closures or item and attribute getters.

Chapter 10. Context Managers

Context managers allow us to simplify code by ensuring that certain operations are performed before and after a particular block of code is executed. The behavior is achieved because context managers define two special methods, `__enter__()` and `__exit__()`, that Python treats specially in the scope of a `with` statement. When a context manager is created in a `with` statement its `__enter__()` method is automatically called, and when the context manager goes out of scope after its `with` statement its `__exit__()` method is automatically called.

We can create our own custom context managers or use predefined ones—as we will see later in this subsection, the file objects returned by the built-in `open()` function are context managers. The syntax for using context managers is this:

```
with expression as variable:
    suite
```

The *`expression`* must be or must produce a context manager object; if the optional `as`*`variable`* part is specified, the variable is set to refer to the object returned by the context manager's `__enter__()` method (and this is often the context manager itself). Because a context manager is guaranteed to execute its "exit" code (even in the face of exceptions), context managers can be used to eliminate the need for `finally` blocks in many situations.

Some of Python's types are context managers—for example, all the file objects that `open()` can return—so we can eliminate `finally` blocks when doing file handling as these equivalent code snippets illustrate (assuming that `process()` is a function defined elsewhere):

```
fh = None
try:
    fh = open(filename)
    for line in fh:
        process(line)
except EnvironmentError as err:
    print(err)
finally:
    if fh is not None:
        fh.close()
```

```
try:
    with open(filename) as fh:
        for line in fh:
            process(line)
except EnvironmentError as err:
    print(err)
```

A file object is a context manager whose exit code always closes the file if it was opened. The exit code is executed whether or not an exception occurs, but in the latter case, the exception is propagated. This ensures that the file gets closed and we still get the chance to handle any errors, in this case by printing a message for the user.

In fact, context managers don't have to propagate exceptions, but not doing so effectively hides any exceptions, and this would almost certainly be a coding error. All the built-in and standard library context managers propagate exceptions.

Sometimes we need to use more than one context manager at the same time. For example:

```
try:
  with open(source) as fin:
    with open(target, "w") as fout:

      for line in fin:
        fout.write(process(line))
  except EnvironmentError as err:
    print(err)
```

Here we read lines from the source file and write processed versions of them to the target file.

Using nested `with` statements can quickly lead to a lot of indentation. Fortunately, the standard library's `contextlib` module provides some additional support for context managers, including the `contextlib.nested()` function which allows two or more context managers to be handled in the same `with` statement rather than having to nest `with` statements. Here is a replacement for the code just shown, but omitting most of the lines that are identical to before:

```
try:
   with contextlib.nested(open(source), open(target, "w")) as (
         fin, fout):
      for line in fin:
```

It isn't only file objects that are context managers. For example, several threading-related classes used for locking are context managers. Context managers can also be used with `decimal.Decimal` numbers; this is useful if we want to perform some calculations with certain settings (such as a particular precision) in effect.

If we want to create a custom context manager we must create a class that provides two methods: `__enter__()` and `__exit__()`. Whenever a `with` statement is used on an instance of such a class, the `__enter__()` method is called and the return value is used for the `as`*`variable`* (or thrown away if there isn't one). When control leaves the scope of the `with` statement the `__exit__()` method is called (with details of an exception if one has occurred passed as arguments).

Suppose we want to perform several operations on a list in an atomic manner—that is, we either want all the operations to be done or none of them so that the resultant list is always in a known state. For example, if we have a list of integers and want to append an integer, delete an integer, and change a couple of integers, all as a single operation, we could write code like this:

```
try:
   with AtomicList(items) as atomic:
      atomic.append(58289)
      del atomic[3]
      atomic[8] = 81738
      atomic[index] = 38172
except (AttributeError, IndexError, ValueError) as err:
   print("no changes applied:", err)
```

If no exception occurs, all the operations are applied to the original list (`items`), but if an exception occurs, no changes are made at all. Here is the code for the `AtomicList` context manager:

```
class AtomicList:

    def __init__(self, alist, shallow_copy=True):
        self.original = alist
        self.shallow_copy = shallow_copy

    def __enter__(self):
        self.modified = (self.original[:] if self.shallow_copy
                    else copy.deepcopy(self.original))
        return self.modified

    def __exit__(self, exc_type, exc_val, exc_tb):
        if exc_type is None:
            self.original[:] = self.modified
```

When the `AtomicList` object is created we keep a reference to the original list and note whether shallow copying is to be used. (Shallow copying is fine for lists of numbers or strings; but for lists that contain lists or other collections, shallow copying is not sufficient.)

Then, when the `AtomicList` context manager object is used in the `with` statement its `__enter__()` method is called. At this point we copy the original list and return the copy so that all the changes can be made on the copy.

Once we reach the end of the `with` statement's scope the `__exit__()` method is called. If no exception occurred the `exc_type` ("exception type") will be `None` and we know that we can safely replace the original list's items with the items from the modified list. (We cannot do `self.original = self.modified` because that would just replace one object reference with another and would not affect the original list at all.) But if an exception occurred, we do nothing to the original list and the modified list is discarded.

The return value of `__exit__()` is used to indicate whether any exception that occurred should be propagated. A `True` value means that we have handled any exception and so no propagation should occur. Normally we always return `False` or something that evaluates to `False` in a Boolean context to allow any exception that occurred to propagate. By not giving an explicit return value, our `__exit__()` returns `None` which evaluates to `False` and correctly causes any exception to propagate.

Potential uses of context managers include ensuring that socket connections and gzipped files are closed, and that threading locks are unlocked.

Chapter 11. Descriptors

Descriptors are classes which provide access control for the attributes of other classes. Any class that implements one or more of the descriptor special methods, `__get__()`, `__set__()`, and `__delete__()`, is called (and can be used as) a descriptor.

The built-in `property()` and `classmethod()` functions are implemented using descriptors. The key to understanding descriptors is that although we create an instance of a descriptor in a class as a class attribute, Python accesses the descriptor through the class's instances.

To make things clear, let's imagine that we have a class whose instances hold some strings. We want to access the strings in the normal way, for example, as a property, but we also want to get an XML-escaped version of the strings whenever we want. One simple solution would be that whenever a string is set we immediately create an XML-escaped copy. But if we had thousands of strings and only ever read the XML version of a few of them, we would be wasting a lot of processing and memory for nothing. So we will create a descriptor that will provide XML-escaped strings on demand without storing them. We will start with the beginning of the client (owner) class, that is, the class that uses the descriptor:

```
class Product:

    __slots__ = ("__name", "__description", "__price")

    name_as_xml = XmlShadow("name")
    description_as_xml = XmlShadow("description")

    def __init__(self, name, description, price):
        self.__name = name
        self.description = description
        self.price = price
```

The only code we have not shown are the properties; the name is a read-only property and the description and price are readable/writable properties, all set up in the usual way. (All the code is in the `XmlShadow.py` file.) We have used the `__slots__` variable to ensure that the class has no `__dict__` and can store only the three specified private attributes; this is not related to or necessary for our use of descriptors. The `name_as_xml` and `description_as_xml` class attributes are set to be instances of the `XmlShadow` descriptor. Although no `Product` object has a `name_as_xml` attribute or a `description_as_xml` attribute, thanks to the descriptor we can write code like this (here quoting from the module's doctests):

```
>>> product = Product("Chisel <3cm>", "Chisel & cap", 45.25)
>>> product.name, product.name_as_xml, product.description_as_xml

('Chisel <3cm>', 'Chisel &lt;3cm&gt;', 'Chisel &amp; cap')
```

This works because when we try to access, for example, the `name_as_xml` attribute, Python finds that the `Product` class has a descriptor with that name, and so uses the descriptor to get the attribute's value. Here's the complete code for the `XmlShadow` descriptor class:

```
class XmlShadow:

    def __init__(self, attribute_name):
        self.attribute_name = attribute_name

    def __get__(self, instance, owner=None):
        return xml.sax.saxutils.escape(
                        getattr(instance, self.attribute_name))
```

When the `name_as_xml` and `description_as_xml` objects are created we pass the name of the `Product` class's corresponding attribute to the `XmlShadow` initializer so that the descriptor knows which attribute to work on. Then, when the `name_as_xml` or `description_as_xml` attribute is looked up, Python calls the descriptor's `__get__()` method. The `self` argument is the instance of the descriptor, the `instance` argument is the `Product` instance (i.e., the product's `self`), and the `owner` argument is the owning class (`Product` in this case). We use the `getattr()` function to retrieve the relevant attribute from the product (in this case the relevant property), and return an XML-escaped version of it.

If the use case was that only a small proportion of the products were accessed for their XML strings, but the strings were often long and the same ones were frequently accessed, we could use a cache. For example:

```
class CachedXmlShadow:

    def __init__(self, attribute_name):
        self.attribute_name = attribute_name
        self.cache = {}

    def __get__(self, instance, owner=None):
        xml_text = self.cache.get(id(instance))
        if xml_text is not None:
            return xml_text
        return self.cache.setdefault(id(instance),
                xml.sax.saxutils.escape(
                        getattr(instance, self.attribute_name)))
```

We store the unique identity of the instance as the key rather than the instance itself because dictionary keys must be hashable (which IDs are), but we don't want to impose that as a requirement on classes that use the `CachedXmlShadow` descriptor. The key is necessary because descriptors are created per class rather than per instance. (The `dict.setdefault()` method conveniently returns the value for the given key, or if no item with that key is present, creates a new item with the given key and value and returns the value.)

Having seen descriptors used to generate data without necessarily storing it, we will now look at a descriptor that can be used to store all of an object's attribute data, with the object not needing to store anything itself. In the example, we will just use a dictionary, but in a more realistic context, the data might be stored in a file or a database. Here's the start of a modified version of the `Point` class that makes use of the descriptor (from the `ExternalStorage.py` file):

```
class Point:

    __slots__ = ()
    x = ExternalStorage("x")
    y = ExternalStorage("y")

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
```

By setting `__slots__` to an empty tuple we ensure that the class cannot store any data attributes at all. When `self.x` is assigned to, Python finds that there is a descriptor with the name “x”, and so uses the descriptor’s `__set__()` method. Here is the complete `ExternalStorage` descriptor class:

```
class ExternalStorage:

    __slots__ = ("attribute_name",)
    __storage = {}

    def __init__(self, attribute_name):
        self.attribute_name = attribute_name

    def __set__(self, instance, value):
        self.__storage[id(instance), self.attribute_name] = value

    def __get__(self, instance, owner=None):
        if instance is None:
            return self
        return self.__storage[id(instance), self.attribute_name]
```

Each `ExternalStorage` object has a single data attribute, `attribute_name`, which holds the name of the owner class’s data attribute. Whenever an attribute is set we store its value in the private class dictionary, `__storage`. Similarly, whenever an attribute is retrieved we get it from the `__storage` dictionary.

As with all descriptor methods, `self` is the instance of the descriptor object and `instance` is the `self` of the object that contains the descriptor, so here `self` is an `ExternalStorage` object and `instance` is a `Point` object.

Although `__storage` is a class attribute, we can access it as `self.__storage` (just as we can call methods using `self.`*`method()`*), because Python will look for it as an instance attribute, and not finding it will then look for it as a class attribute. The one (theoretical) disadvantage of this approach is that if we have a class attribute and an instance attribute with the same name, one would hide the other. (If this were really a problem we could always refer to the class attribute using the class, that is, `ExternalStorage.__storage`. Although hard-coding the class does not play well with subclassing in general, it doesn’t really matter for private attributes since Python name-mangles the class name into them anyway.)

The implementation of the `__get__()` special method is slightly more sophisticated than before because we provide a means by which the `ExternalStorage` instance itself can be accessed. For example, if we have `p = Point(3, 4)`, we can access the *x*-coordinate with `p.x`, and we can access the `ExternalStorage` object that holds all the `x`s with `Point.x`.

To complete our coverage of descriptors we will create the `Property` descriptor that mimics the behavior of the built-in `property()` function, at least for setters and getters. The code is in `Property.py`. Here is the complete `NameAndExtension` class that makes use of it:

```
class NameAndExtension:

    def __init__(self, name, extension):
        self.__name = name
        self.extension = extension

    @Property            # Uses the custom Property descriptor
    def name(self):
        return self.__name

    @Property            # Uses the custom Property descriptor
    def extension(self):
        return self.__extension

    @extension.setter       # Uses the custom Property descriptor
    def extension(self, extension):
        self.__extension = extension
```

The usage is just the same as for the built-in `@property` decorator and for the `@`*`propertyName`*`.setter` decorator. Here is the start of the `Property` descriptor's implementation:

```
class Property:

    def __init__(self, getter, setter=None):
        self.__getter = getter
        self.__setter = setter
        self.__name__ = getter.__name__
```

The class's initializer takes one or two functions as arguments. If it is used as a decorator, it will get just the decorated function and this becomes the getter, while the setter is set to `None`. We use the getter's name as the property's name. So for each property, we have a getter, possibly a setter, and a name.

```
def __get__(self, instance, owner=None):
    if instance is None:
        return self
    return self.__getter(instance)
```

When a property is accessed we return the result of calling the getter function where we have passed the instance as its first parameter. At first sight, `self.__getter()` looks like a method call, but it is not. In fact, `self.__getter` is an attribute, one that happens to hold an object reference to a method that was passed in. So what happens is that first we retrieve the attribute (`self.__getter`), and then we call it as a function `()`. And because it is called as a function rather than as a method we must pass in the relevant `self` object explicitly ourselves. And in the case of a descriptor the `self` object (from the class that is using the descriptor) is called `instance` (since `self` is the descriptor object). The same applies to the `__set__()` method.

```
def __set__(self, instance, value):
    if self.__setter is None:
        raise AttributeError("'{0}' is read-only".format(
                    self.__name__))
    return self.__setter(instance, value)
```

If no setter has been specified, we raise an `AttributeError`; otherwise, we call the setter with the instance and the new value.

```
def setter(self, setter):
    self.__setter = setter
    return self.__setter
```

This method is called when the interpreter reaches, for example, `@extension.setter`, with the function it decorates as its `setter` argument. It stores the setter method it has been given (which can now be used in the `__set__()` method), and returns the setter, since decorators should return the function or method they decorate.

We have now looked at three quite different uses of descriptors. Descriptors are a very powerful and flexible feature that can be used to do lots of under-the-hood work while appearing to be simple attributes in their client (owner) class.

Chapter 12. Class Decorators

Just as we can create decorators for functions and methods, we can also create decorators for entire classes. Class decorators take a class object (the result of the `class` statement), and should return a class—normally a modified version of the class they decorate. In this subsection we will study two class decorators to see how they can be implemented.

In the book's examples there is a `SortedList.py` module that defines the `SortedList` custom collection class. This class aggregates a plain list as the private attribute `self.__list`, and eight of the `SortedList` class's methods simply pass on their work to the private attribute. For example, here are how the `SortedList.clear()` and `SortedList.pop()` methods are implemented:

```
def clear(self):
    self.__list = []

def pop(self, index=-1):
    return self.__list.pop(index)
```

There is nothing we can do about the `clear()` method since there is no corresponding method for the `list` type, but for `pop()`, and the other six methods that `SortedList` delegates, we can simply call the `list` class's corresponding method. This can be done by using the `@delegate` class decorator from the book's `Util` module. Here is the start of a new version of the `SortedList` class:

```
@Util.delegate("__list", ("pop", "__delitem__", "__getitem__",
               "__iter__", "__reversed__", "__len__", "__str__"))
class SortedList:
```

The first argument is the name of the attribute to delegate to, and the second argument is a sequence of one or more methods that we want the `delegate()` decorator to implement for us so that we don't have to do the work ourselves. The `SortedList` class in the `SortedListDelegate.py` file uses this approach and therefore does not have any code for the methods listed, even though it fully supports them. Here is the class decorator that implements the methods for us:

```
def delegate(attribute_name, method_names):
    def decorator(cls):
        nonlocal attribute_name

        if attribute_name.startswith("__"):
            attribute_name = "_" + cls.__name__ + attribute_name
        for name in method_names:
            setattr(cls, name, eval("lambda self, *a, **kw: "
                            "self.{0}.{1}(*a, **kw)".format(
                            attribute_name, name)))
        return cls
    return decorator
```

We could not use a plain decorator because we want to pass arguments to the decorator, so we have instead created a function that takes our arguments and that returns a class decorator. The decorator itself takes a single argument, a class (just as a function decorator takes a single function or method as its argument).

We must use `nonlocal` so that the nested function uses the `attribute_name` from the outer scope rather than attempting to use one from its own scope. And we must be able to correct the attribute name if necessary to take account of the name mangling of private attributes. The decorator's behavior is quite simple: It iterates over all the method names that the `delegate()` function has been given, and for each one creates a new method which it sets as an attribute on the class with the given method name.

We have used `eval()` to create each of the delegated methods since it can be used to execute a single statement, and a `lambda` statement produces a method or function. For example, the code executed to produce the `pop()` method is:

```
lambda self, *a, **kw: self._SortedList__list.pop(*a, **kw)
```

We use the `*` and `**` argument forms to allow for any arguments even though the methods being delegated to have specific argument lists. For example, `list.pop()` accepts a single index position (or nothing, in which case it defaults to the last item). This is okay because if the wrong number or kinds of arguments are passed, the `list` method that is called to do the work will raise an appropriate exception.

The second class decorator we will review is used with the `FuzzyBool.py` module from the book's examples. This module defines the `FuzzyBool` class, and although the implementation only defines two comparison special methods, `__lt__()` and `__eq__()` (for < and ==), all the other comparison methods are supported automatically thanks to the use of a class decorator:

```
@Util.complete_comparisons
class FuzzyBool:
```

The other four comparison operators were provided by the `complete_comparisons()` class decorator. Given a class that defines only < (or < and ==), the decorator produces the missing comparison operators by using the following logical equivalences:

$$
\begin{aligned}
x = y &\Leftrightarrow \neg(x < y \vee y < x)\\
x \neq y &\Leftrightarrow \neg(x = y)\\
x > y &\Leftrightarrow y < x\\
x \leq y &\Leftrightarrow \neg(y < x)\\
x \geq y &\Leftrightarrow \neg(x < y)
\end{aligned}
$$

If the class to be decorated has < and ==, the decorator will use them both, falling back to doing everything in terms of < if that is the only operator supplied. (In fact, Python automatically produces > if < is supplied, != if == is supplied, and >= if <= is supplied, so it is sufficient to just implement the three operators <, <=, and == and to leave Python to infer the others. However, using the class decorator reduces the minimum that we must implement to just <. This is convenient, and also ensures that all the comparison operators use the same consistent logic.)

```
def complete_comparisons(cls):
    assert cls.__lt__ is not object.__lt__, (
            "{0} must define < and ideally ==".format(cls.__name__))
    if cls.__eq__ is object.__eq__:
        cls.__eq__ = lambda self, other: (not
                (cls.__lt__(self, other) or cls.__lt__(other, self)))
    cls.__ne__ = lambda self, other: not cls.__eq__(self, other)
    cls.__gt__ = lambda self, other: cls.__lt__(other, self)
    cls.__le__ = lambda self, other: not cls.__lt__(other, self)
    cls.__ge__ = lambda self, other: not cls.__lt__(self, other)
    return cls
```

One problem that the decorator faces is that class `object` from which every other class is ultimately derived defines all six comparison operators, all of which raise a `TypeError` exception if used. So we need to know whether < and == have been reimplemented (and are therefore usable). This can easily be done by comparing the relevant special methods in the class being decorated with those in `object`.

If the decorated class does not have a custom < the assertion fails because that is the decorator's minimum requirement. And if there is a custom == we use it; otherwise, we create one. Then all the other methods are created and the decorated class, now with all six comparison methods, is returned.

Using class decorators is probably the simplest and most direct way of changing classes. Another approach is to use metaclasses, a topic we will cover later in this short cut.

Meta-classes

☞54

Chapter 13. Abstract Base Classes

An abstract base class (ABC) is a class that cannot be used to create objects. Instead, the purpose of such classes is to define interfaces, that is, to in effect list the methods and properties that classes that inherit the abstract base class must provide. This is useful because we can use an abstract base class as a kind of promise—a promise that any derived class will provide the methods and properties that the abstract base class specifies.[*]

[*] Python's abstract base classes are described in PEP 3119 (www.python.org/dev/peps/pep-3119), which also includes a very useful rationale and is well worth reading.

Abstract base classes are classes that have at least one abstract method or property. Abstract methods can be defined with no implementation (i.e., their suite is `pass`, or if we want to force reimplementation in a subclass, `raise NotImplementedError()`), or with an actual (concrete) implementation that can be invoked from subclasses, for example, when there is a common case. They can also have other concrete (i.e., nonabstract) methods and properties.

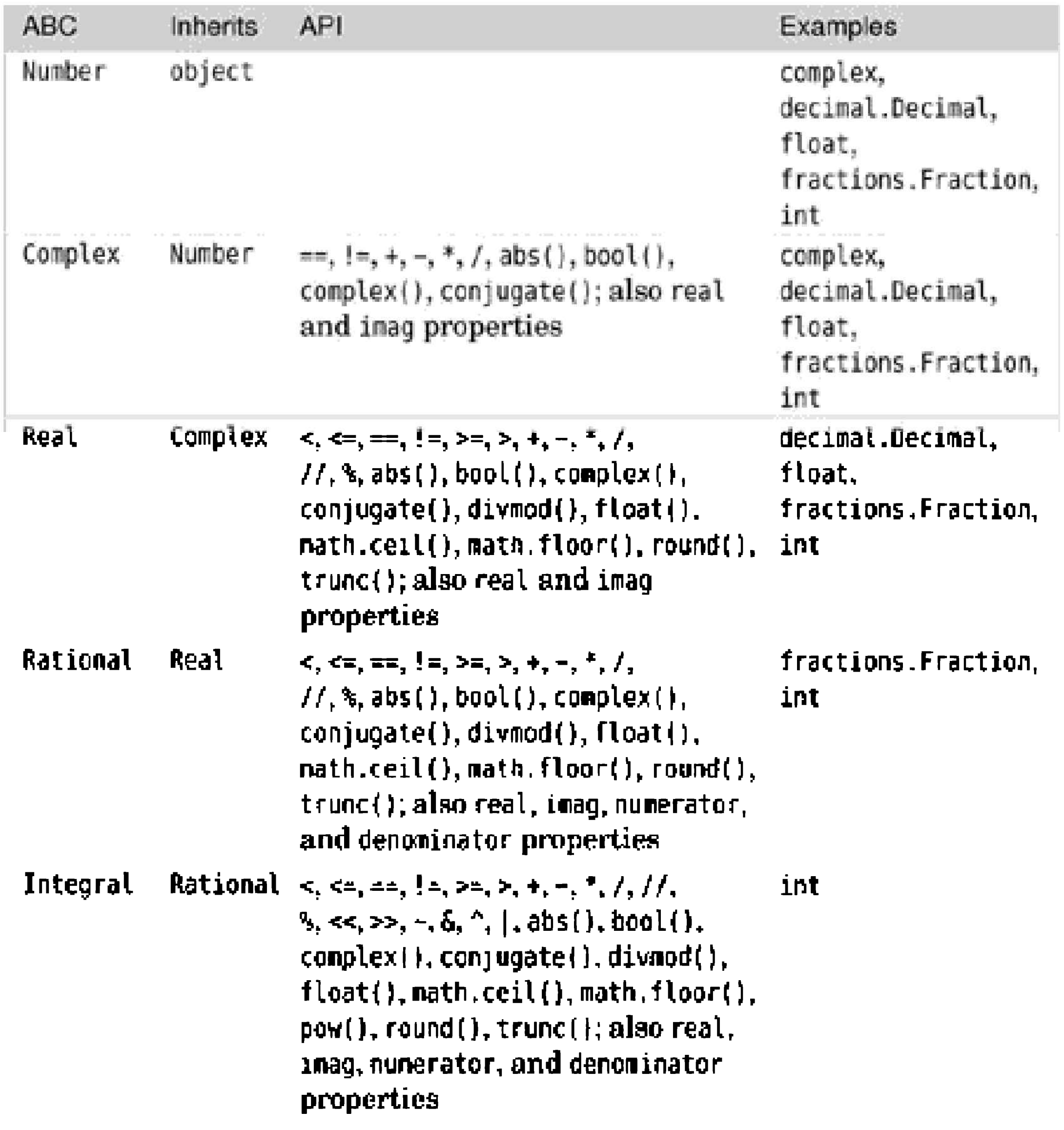
Classes that derive from an ABC can be used to create instances only if they reimplement all the abstract methods and abstract properties they have inherited. For those abstract methods that have concrete implementations (even if it is only `pass`), the derived class could simply use `super()` to use the ABC's version. Any concrete methods or properties are available through inheritance as usual. All ABCs must have a metaclass of `abc.ABCMeta` (from the `abc` module), or from one of its subclasses. We cover metaclasses a bit further on.

Meta-classes

☞54

Python provides two groups of abstract base classes, one in the `collections` module and the other in the `numbers` module. They allow us to ask questions about an object; for example, given a variable *x*, we can see whether it is a sequence using `isinstance(x, collections.MutableSequence)` or whether it is a whole number using `isinstance(x, numbers.Integral)`. This is particularly useful in view of Python's dynamic typing where we don't necessarily know (or care) what an object's type is, but want to know whether it supports the operations we want to apply to it. The numeric and collection ABCs are listed in Tables 3 and 4. The other major ABC is `io.IOBase` from which all the file and stream-handling classes derive.

Table 3. The Numbers Module's Abstract Base Classes



ABC	Inherits	API	Examples
Number	object		complex, decimal.Decimal, float, fractions.Fraction, int
Complex	Number	==, !=, +, -, *, /, abs(), bool(), complex(), conjugate(); also real and imag properties	complex, decimal.Decimal, float, fractions.Fraction, int
Real	Complex	<, <=, ==, !=, >=, >, +, -, *, /, //, %, abs(), bool(), complex(), conjugate(), divmod(), float(), math.ceil(), math.floor(), round(), trunc(); also real and imag properties	decimal.Decimal, float, fractions.Fraction, int
Rational	Real	<, <=, ==, !=, >=, >, +, -, *, /, //, %, abs(), bool(), complex(), conjugate(), divmod(), float(), math.ceil(), math.floor(), round(), trunc(); also real, imag, numerator, and denominator properties	fractions.Fraction, int
Integral	Rational	<, <=, ==, !=, >=, >, +, -, *, /, //, %, <<, >>, ~, &, ^, |, abs(), bool(), complex(), conjugate(), divmod(), float(), math.ceil(), math.floor(), pow(), round(), trunc(); also real, imag, numerator, and denominator properties	int

Table 4. The Collections Module's Main Abstract Base Classes

ABC	Inherits	API	Examples
`Callable`	`object`	`()`	All functions, methods, and `lambdas`
`Container`	`object`	`in`	`bytearray`, `bytes`, `dict`, `frozenset`, `list`, `set`, `str`, `tuple`
`Hashable`	`object`	`hash()`	`bytes`, `frozenset`, `str`, `tuple`
`Iterable`	`object`	`iter()`	`bytearray`, `bytes`, `collections.deque`, `dict`, `frozenset`, `list`, `set`, `str`, `tuple`
`Iterator`	`Iterable`	`iter()`, `next()`	
`Sized`	`object`	`len()`	`bytearray`, `bytes`, `collections.deque`, `dict`, `frozenset`, `list`, `set`, `str`, `tuple`
`Mapping`	`Container`, `Iterable`, `Sized`	`==`, `!=`, `[]`, `len()`, `iter()`, `in`, `get()`, `items()`, `keys()`, `values()`	`dict`

Mutable-Mapping	Mapping	==, !=, [], del, len(), iter(), in, clear(), get(), items(), keys(), pop(), popitem(), setdefault(), update(), values()	dict
Sequence	Container, Iterable, Sized	[], len(), iter(), reversed(), in, count(), index()	bytearray, bytes, list, str, tuple
Mutable-Sequence	Container, Iterable, Sized	[], +=, del, len(), iter(), reversed(), in, append(), count(), extend(), index(), insert(), pop(), remove(), reverse()	bytearray, list
Set	Container, Iterable, Sized	<, <=, ==, !=, =>, >, &, |, ^, len(), iter(), in, isdisjoint()	frozenset, set
MutableSet	Set	<, <=, ==, !=, =>, >, &, |, ^, &=, |=, ^=, -=, len(), iter(), in, add(), clear(), discard(), isdisjoint(), pop(), remove()	set

To fully integrate our own custom numeric and collection classes we ought to make them fit in with the standard ABCs. For example, the `SortedList` class is a sequence, but as it stands, `isinstance(L, collections.Sequence)` returns `False` if `L` is a `SortedList`. One easy way to fix this is to inherit the relevant ABC:

```
class SortedList(collections.Sequence):
```

By making `collections.Sequence` the base class, the `isinstance()` test will now return `True`. Furthermore, we will be required to implement `__init__()` (or `__new__()`), `__getitem__()`, and `__len__()` (which we do). The `collections.Sequence` ABC also provides concrete (i.e., nonabstract) implementations for `__contains__()`, `__iter__()`, `__reversed__()`, `count()`, and `index()`. In the case of `SortedList`, we reimplement them all, but we could have used the ABC versions if we wanted to, simply by not reimplementing them. We cannot make `SortedList` a subclass of `collections.MutableSequence` even though the list is mutable because `SortedList` does not have all the methods that a collections.MutableSequence must provide, such as __setitem__() and append(). (The code for this SortedList is in SortedListAbc.py. We will see an alternative approach to making a SortedList into a collections.Sequence in the Metaclasses subsection.)

Meta-classes

☞54

Now that we have seen how to make a custom class fit in with the standard ABCs, we will turn to another use of ABCs: to provide an interface promise for our own custom classes. We will look at three rather different examples to cover different aspects of creating and using ABCs.

We will start with a very simple example that shows how to handle read-able/ writable properties. The class is used to represent domestic appliances. Every appliance that is created must have a read-only model string and a read-able/ writable price. We also want to ensure that the ABC's __init__() is reimplemented. Here's the ABC (from Appliance.py); we have not shown the import abc statement which is needed for the abstractmethod() and abstractproperty() functions, both of which can be used as decorators:

```
class Appliance(metaclass=abc.ABCMeta):

    @abc.abstractmethod
    def __init__(self, model, price):
        self.__model = model
        self.price = price

    def get_price(self):
        return self.__price
    def set_price(self, price):
        self.__price = price

    price = abc.abstractproperty(get_price, set_price)

    @property
    def model(self):
        return self.__model
```

We have set the class's metaclass to be `abc.ABCMeta` since this is a requirement for ABCs; any `abc.ABCMeta` subclass can be used instead, of course. We have made `__init__()` an abstract method to ensure that it is reimplemented, and we have also provided an implementation which we expect (but can't force) inheritors to call. To make an abstract readable/writable property we cannot use decorator syntax; also we have not used private names for the getter and setter since doing so would be inconvenient for subclasses. The `model` property is not abstract, so subclasses don't need to reimplement it. No `Appliance` objects can be created because the class contains abstract attributes. Here is an example subclass:

```
class Cooker(Appliance):

  def __init__(self, model, price, fuel):
    super().__init__(model, price)
    self.fuel = fuel

    price = property(lambda self: super().price,
             lambda self, price: super().set_price(price))
```

The `Cooker` class must reimplement the `__init__()` method and the `price` property. For the property we have just passed on all the work to the base class. The `model` read-only property is inherited. We could create many more classes based on `Appliance`, such as `Fridge`, `Toaster`, and so on.

The next ABC we will look at is even shorter; it is an ABC for text-filtering functors (in file `TextFilter.py`):

```
class TextFilter(metaclass=abc.ABCMeta):

  @abc.abstractproperty
  def is_transformer(self):
    raise NotImplementedError()

  @abc.abstractmethod
  def __call__(self):
    raise NotImplementedError()
```

The `TextFilter` ABC provides no functionality at all; it exists purely to define an interface, in this case an `is_transformer` read-only property and a `__call__()` method, that all its subclasses must provide. Since the abstract property and method have no implementations we don't want subclasses to call them, so instead of using an innocuous `pass` statement we raise an exception if they are used (e.g., via a `super()` call).

Here is one simple subclass:

```
class CharCounter(TextFilter):

    @property
    def is_transformer(self):
        return False

    def __call__(self, text, chars):
        count = 0
        for c in text:
            if c in chars:
                count += 1
        return count
```

This text filter is not a transformer because rather than transforming the text it is given, it simply returns a count of the specified characters that occur in the text. Here is an example of use:

```
vowel_counter = CharCounter()

vowel_counter("dog fish and cat fish", "aeiou")     # returns: 5
```

Two other text filters are provided, both of which are transformers: `RunLength-Encode` and `RunLengthDecode`. Here is how they are used:

```
rle_encoder = RunLengthEncode()
rle_text = rle_encoder(text)
...
rle_decoder = RunLengthDecode()
original_text = rle_decoder(rle_text)
```

The run length encoder converts a string into UTF-8 encoded bytes, and replaces `0x00` bytes with the sequence `0x00`, `0x01`, `0x00`, and any sequence of three to 255 repeated bytes with the sequence `0x00`, *count*, *byte*. If the string has lots of runs of four or more identical consecutive characters this can produce a shorter byte string than the raw UTF-8 encoded bytes. The run length decoder takes a run length encoded byte string and returns the original string. Here is the start of the `RunLengthDecode` class:

```
class RunLengthDecode(TextFilter):

    @property
    def is_transformer(self):
        return True

    def __call__(self, rle_bytes):
        ...
```

We have omitted the body of the `__call__()` method, although it is in the source that accompanies this book. The `RunLengthEncode` class has exactly the same structure.

The last ABC we will look at provides an Application Programming Interface (API) and a default implementation for an undo mechanism. Here is the complete ABC (from file `Abstract.py`):

```
class Undo(metaclass=abc.ABCMeta):


    @abc.abstractmethod
    def __init__(self):
        self.__undos = []

    @abc.abstractproperty
    def can_undo(self):
        return bool(self.__undos)

    @abc.abstractmethod


    def undo(self):
        assert self.__undos, "nothing left to undo"
        self.__undos.pop()(self)

    def add_undo(self, undo):
        self.__undos.append(undo)
```

The `__init__()` and `undo()` methods must be reimplemented since they are both abstract; and so must the read-only `can_undo` property. Subclasses don’t have to reimplement the `add_undo()` method, although they are free to do so. The `undo()` method is slightly subtle. The `self.__undos` list is expected to hold object references to methods. Each method must cause the corresponding action to be undone if it is called—this will be clearer when we look at an `Undo` subclass in a moment. So to perform an undo we pop the last undo method off the `self.__undos` list, and then call the method as a function, passing `self` as an argument. (We must pass `self` because the method is being called as a function and not as a method.)

Here is the beginning of the `Stack` class; it inherits `Undo`, so any actions performed on it can be undone by calling `Stack.undo()` with no arguments:

```
class Stack(Undo):

    def __init__(self):
        super().__init__()
        self.__stack = []

    @property
    def can_undo(self):
        return super().can_undo

    def undo(self):
        super().undo()

    def push(self, item):
        self.__stack.append(item)
        self.add_undo(lambda self: self.__stack.pop())

    def pop(self):
        item = self.__stack.pop()
        self.add_undo(lambda self: self.__stack.append(item))
        return item
```

We have omitted `Stack.top()` and `Stack.__str__()` since neither adds anything new and neither interacts with the `Undo` base class. For the `can_undo` property and the `undo()` method, we simply pass on the work to the base class. If these two were not abstract we would not need to reimplement them at all and the same effect would be achieved; but in this case we wanted to force subclasses to reimplement them to encourage undo to be taken account of in the subclass. For `push()` and `pop()` we perform the operation and also add a function to the undo list which will undo the operation that has just been performed.

Abstract base classes are most useful in large-scale programs, libraries, and application frameworks, where they can help ensure that irrespective of implementation details or author, classes can work cooperatively together because they provide the APIs that their ABCs specify.

Chapter 14. Multiple Inheritance

Multiple inheritance is where one class inherits from two or more other classes. Although Python (and, for example, C++) fully supports multiple inheritance, some languages—most notably, Java—don't allow it. One problem is that multiple inheritance can lead to the same class being inherited more than once (e.g., if two of the base classes inherit from the same class), and this means that the version of a method that is called, if it is not in the subclass but is in two or more of the base classes (or their base classes, etc.), depends on the method resolution order, which potentially makes classes that use multiple inheritance somewhat fragile.

Multiple inheritance can generally be avoided by using single inheritance (one base class), and setting a metaclass if we want to support an additional API, since as we will see in the next subsection, a metaclass can be used to give the promise of an API without actually inheriting any methods or data attributes. An alternative is to use multiple inheritance with one concrete class and one or more abstract base classes for additional APIs. And another alternative is to use single inheritance and aggregate instances of other classes.

Nonetheless, in some cases, multiple inheritance can provide a very convenient solution. For example, suppose we want to create a new version of the `Stack` class from the previous subsection, but want the class to support loading and saving using a pickle. We might well want to add the loading and saving functionality to several classes, so we will implement it in a class of its own:

```
class LoadSave:

    def __init__(self, filename, *attribute_names):
        self.filename = filename
        self.__attribute_names = []
        for name in attribute_names:
            if name.startswith("__"):
                name = "_" + self.__class__.__name__ + name
            self.__attribute_names.append(name)

   def save(self):
       with open(self.filename, "wb") as fh:
           data = []
           for name in self.__attribute_names:
            data.append(getattr(self, name))
        pickle.dump(data, fh, pickle.HIGHEST_PROTOCOL)

def load(self):
    with open(self.filename, "rb") as fh:
        data = pickle.load(fh)
        for name, value in zip(self.__attribute_names, data):
            setattr(self, name, value)
```

The class has two attributes: `filename`, which is public and can be changed at any time, and `__attribute_names`, which is fixed and can be set only when the instance is created. The `save()` method iterates over all the attribute names and creates a list called `data` that holds the value of each attribute to be saved; it then saves the data into a pickle. The `with` statement ensures that the file is closed if it was successfully opened, and any file or pickle exceptions are passed up to the caller. The `load()` method iterates over the attribute names and the corresponding data items that have been loaded and sets each attribute to its loaded value.

Here is the start of the `FileStack` class that multiply-inherits the `Undo` class from the previous subsection and this subsection's `LoadSave` class:

```
class FileStack(Undo, LoadSave):

    def __init__(self, filename):
        Undo.__init__(self)
        LoadSave.__init__(self, filename, "__stack")
        self.__stack = []

    def load(self):
        super().load()
        self.clear()
```

The rest of the class is just the same as the `Stack` class, so we have not reproduced it here. Instead of using `super()` in the `__init__()` method we must specify the base classes that we initialize since `super()` cannot guess our intentions. For the `LoadSave` initialization we pass the filename to use and also the names of the attributes we want saved; in this case just one, the private `__stack`. (We don't want to save the `__undos`; and nor could we in this case since it is a list of methods and is therefore unpicklable.)

The `FileStack` class has all the `Undo` methods, and also the `LoadSave` class's `save()` and `load()` methods. We have not reimplemented `save()` since it works fine, but for `load()` we must clear the undo stack after loading. This is necessary because we might do a save, then do various changes, and then a load. The load wipes out what went before, so any undos no longer make sense. The original `Undo` class did not have a `clear()` method, so we had to add one:

```
def clear(self):         # In class Undo
    self.__undos = []
```

In the `Stack.load()` method we have used `super()` to call `LoadSave.load()` because there is no `Undo.load()` method to cause ambiguity. If both base classes had had a `load()` method, the one that would get called would depend on Python's method resolution order. We prefer to use `super()` only when there is no ambiguity, and to use the appropriate base name otherwise, so we never rely on the method resolution order. For the `self.clear()` call, again there is no ambiguity since only the `Undo` class has a `clear()` method, and we don't need to use `super()` since (unlike `load()`) `FileStack` does not have a `clear()` method.

What would happen if, later on, a `clear()` method was added to the `FileStack` class? It would break the `load()` method. One solution would be to call `super().clear()` inside `load()` instead of plain `self.clear()`. This would result in the first super-class's `clear()` method that was found being used. To protect against such problems we could make it a policy to use hard-coded base classes when using multiple inheritance (in this example, calling `Undo.clear(self)`). Or we could avoid multiple inheritance altogether and use aggregation, for example, inheriting the `Undo` class and creating a `LoadSave` class designed for aggregation.

What multiple inheritance has given us here is a mixture of two rather different classes, without the need to implement any of the undo or the loading and saving ourselves, relying instead on the functionality provided by the base classes. This can be very convenient and works especially well when the inherited classes have no overlapping APIs.

Chapter 15. Metaclasses

A metaclass is to a class what a class is to an instance; that is, a metaclass is used to create classes, just as classes are used to create instances. And just as we can ask whether an instance belongs to a class by using `isinstance()`, we can ask whether a class object (such as `dict`, `int`, or `SortedList`) inherits another class using `issubclass()`.

The simplest use of metaclasses is to make custom classes fit into Python's standard ABC hierarchy. For example, to make `SortedList` a `collections. Sequence`, instead of inheriting the ABC (as we showed earlier), we can simply register the `SortedList` as a `collections.Sequence`:

```
class SortedList:
    ...
collections.Sequence.register(SortedList)
```

After the class is defined normally, we register it with the `collections.Sequence` ABC. Registering a class like this makes it a *virtual subclass*.[*] A virtual subclass reports that it is a subclass of the class or classes it is registered with (e.g., using `isinstance()` or `issubclass()`), but does not inherit any data or methods from any of the classes it is registered with.

Registering a class like this provides a promise that the class provides the API of the classes it is registered with, but does not provide any guarantee that it will honor its promise. One use of metaclasses is to provide both a promise and a guarantee about a class's API. Another use is to modify a class in some way (like a class decorator does). And of course, metaclasses can be used for both purposes at the same time.

Suppose we want to create a group of classes that all provide `load()` and `save()` methods. We can do this by creating a class that when used as a metaclass, checks that these methods are present:

```
class LoadableSaveable(type):

    def __init__(cls, classname, bases, dictionary):
        super().__init__(classname, bases, dictionary)
        assert hasattr(cls, "load") and \
            isinstance(getattr(cls, "load"),
                  collections.Callable), ("class '" +
            classname + "' must provide a load() method")
        assert hasattr(cls, "save") and \
            isinstance(getattr(cls, "save"),
                  collections.Callable), ("class '" +
            classname + "' must provide a save() method")
```

Classes that are to serve as metaclasses must inherit from the ultimate metaclass base class, `type`, or one of its subclasses.

Note that this class is called when *classes* that use it are instantiated, in all probability not very often, so the runtime cost is extremely low. Notice also that we must perform the checks after the class has been created (using the `super()` call), since only then will the class's attributes be available in the class itself. (The attributes are in the dictionary, but we prefer to work on the actual initialized class when doing checks.)

`collections` ABCs

47

We could have checked that the `load` and `save` attributes are callable using `hasattr()` to check that they have the `__call__` attribute, but we prefer to check whether they are instances of `collections.Callable` instead. The `collections.Callable` abstract base class provides the promise (but no guarantee) that instances of its subclasses (or virtual subclasses) are callable.

Once the class has been created (using `type.__new__()` or a reimplementation of `__new__()`), the metaclass is initialized by calling its `__init__()` method. The arguments given to `__init__()` are `cls`, the class that's just been created; `classname`, the class's name (also available from `cls.__name__`); `bases`, a list of the class's base classes (excluding `object`, and therefore possibly empty); and `dictionary` that holds the attributes that became class attributes when the `cls` class was created, unless we intervened in a reimplementation of the meta-class's `__new__()` method.

Here are a couple of interactive examples that show what happens when we create classes using the `LoadableSaveable` metaclass:

```
>>> class Bad(metaclass=Meta.LoadableSaveable):
...     def some_method(self): pass
Traceback (most recent call last):
...
AssertionError: class 'Bad' must provide a load() method
```

The metaclass specifies that classes using it must provide certain methods, and when they don't, as in this case, an `AssertionError` exception is raised.

```
>>> class Good(metaclass=Meta.LoadableSaveable):
...    def load(self): pass
...    def save(self): pass
>>> g = Good()
```

The `Good` class honors the metaclass's API requirements, even if it doesn't meet our informal expectations of how it should behave.

We can also use metaclasses to change the classes that use them. If the change involves the name, base classes, or dictionary of the class being created (e.g., its slots), then we need to reimplement the metaclass's `__new__()` method; but for other changes, such as adding methods or data attributes, reimplemeting `__init__()` is sufficient, although this can also be done in `__new__()`. We will now look at a metaclass that modifies the classes it is used with purely through its `__new__()` method.

As an alternative to using the `@property` and `@`*`name`*`.setter` decorators, we could create classes where we use a simple naming convention to identify properties. For example, if a class has methods of the form `get_`*`name`*`()` and `set_`*`name`*`()`, we would expect the class to have a private `__`*`name`* property accessed using *`instance.name`* for getting and setting. This can all be done using a metaclass. Here is an example of a class that uses this convention:

```
class Product(metaclass=AutoSlotProperties):

   def __init__(self, barcode, description):
      self.__barcode = barcode
      self.description = description

   def get_barcode(self):
      return self.__barcode
```

```
    def get_description(self):
        return self.__description

    def set_description(self, description):
        if description is None or len(description) < 3:
            self.__description = "<Invalid Description>"
        else:
            self.__description = description
```

We must assign to the private `__barcode` property in the initializer since there is no setter for it; another consequence of this is that `barcode` is a read-only property. On the other hand, `description` is a readable/writable property. Here are some examples of interactive use:

```
>>> product = Product("101110110", "8mm Stapler")
>>> product.barcode, product.description
('101110110', '8mm Stapler')
>>> product.description = "8mm Stapler (long)"
>>> product.barcode, product.description
('101110110', '8mm Stapler (long)')
```

If we attempt to assign to the bar code an `AttributeError` exception is raised with the error text "can't set attribute".

If we look at the `Product` class's attributes (e.g., using `dir()`), the only public ones to be found are `barcode` and `description`. The `get_name()` and `set_name()` methods are no longer there—they have been replaced with the *name* property. And the variables holding the bar code and description are also private (`__bar-code` and `__description`), and have been added as slots to minimize the class's memory use. This is all done by the `AutoSlotProperties` metaclass which is implemented in a single method:

```
class AutoSlotProperties(type):

    def __new__(mcl, classname, bases, dictionary):
        slots = list(dictionary.get("__slots__", []))
        for getter_name in [key for key in dictionary
                        if key.startswith("get_")]:

        if isinstance(dictionary[getter_name],
                    collections.Callable):
            name = getter_name[4:]
            slots.append("__" + name)
            getter = dictionary.pop(getter_name)
            setter_name = "set_" + name
            setter = dictionary.get(setter_name, None)
            if (setter is not None and
                isinstance(setter, collections.Callable)):
                del dictionary[setter_name]
            dictionary[name] = property(getter, setter)
    dictionary["__slots__"] = tuple(slots)
    return super().__new__(mcl, classname, bases, dictionary)
```

A metaclass's `__new__()` class method is called with the metaclass, and the class name, base classes, and dictionary of the class that is to be created. We must use a reimplementation of `__new__()` rather than `__init__()` because we want to change the dictionary before the class is created.

We begin by copying the `__slots__` collection, creating an empty one if none is present, and making sure we have a list rather than a tuple so that we can modify it. For every attribute in the dictionary we pick out those that begin with `"get_"` and that are callable, that is, those that are getter methods. For each getter we add a private name to the slots to store the corresponding data; for example, given getter `get_`*`name`*`()` we add `__`*`name`* to the slots. We then take a reference to the getter and delete it from the dictionary under its original name (this is done in one go using `dict.pop()`). We do the same for the setter if one is present, and then we create a new dictionary item with the desired property name as its key; for example, if the getter is `get_`*`name`*`()` the property name is *`name`*. We set the item's value to be a property with the getter and setter (which might be `None`) that we have found and removed from the dictionary.

At the end we replace the original slots with the modified slots list which has a private slot for each property that was added, and call on the base class to actually create the class, but using our modified dictionary. Note that in this case we must pass the metaclass explicitly in the `super()` call; this is always the case for calls to `__new__()` because it is a class method and not an instance method.

For this example we didn't need to write an `__init__()` method because we have done all the work in `__new__()`, but it is perfectly possible to reimplement both `__new__()` and `__init__()` doing different work in each.

If we consider hand-cranked drills to be analogous to aggregation and inheritance and electric drills the analog of decorators and descriptors, then meta-classes are at the laser beam end of the scale when it comes to power and versatility. Metaclasses are the last tool to reach for rather than the first, except perhaps for application framework developers who need to provide powerful facilities to their users without making the users go through hoops to realize the benefits on offer.

---

[*] In Python terminology, *virtual* does not mean the same thing as it does in C++ terminology.

Chapter 16. Functional-Style Programming

Functional-style programming is an approach to programming where computations are built up from combining functions that don't modify their arguments and that don't refer to or change the program's state, and that provide their results as return values. One strong appeal of this kind of programming is that (in theory), it is much easier to develop functions in isolation and to debug functional programs. This is helped by the fact that functional programs don't have state changes, so it is possible to reason about their functions mathematically.

Three concepts that are strongly associated with functional programming are *mapping*, *filtering*, and *reducing*. Mapping involves taking a function and an iterable and producing a new iterable (or a list) where each item is the result of calling the function on the corresponding item in the original iterable. This is supported by the built-in `map()` function, for example:

```
list(map(lambda x: x ** 2, [1, 2, 3, 4]))  # returns: [1, 4, 9, 16]
```

The `map()` function takes a function and an iterable as its arguments and for efficiency it returns an iterator rather than a list. Here we forced a list to be created to make the result clearer:

```
[x ** 2 for x in [1, 2, 3, 4]]          # returns: [1, 4, 9, 16]
```

A generator expression can often be used in place of `map()`. Here we have used a list comprehension to avoid the need to use `list()`; to make it a generator we just have to change the outer brackets to parentheses.

Filtering involves taking a function and an iterable and producing a new iterable where each item is from the original iterable—providing the function returns `True` when called on the item. The built-in `filter()` function supports this:

```
list(filter(lambda x: x > 0, [1, -2, 3, -4])) # returns: [1, 3]
```

The `filter()` function takes a function and an iterable as its arguments and returns an iterator.

```
[x for x in [1, -2, 3, -4] if x > 0]        # returns: [1, 3]
```

The `filter()` function can always be replaced with a generator expression or with a list comprehension.

Reducing involves taking a function and an iterable and producing a single result value. The way this works is that the function is called on the iterable's first two values, then on the computed result and the third value, then on the computed result and the fourth value, and so on, until all the values have been used. The `functools` module's `functools.reduce()` function supports this. Here are two lines of code that do the same computation:

```
functools.reduce(lambda x, y: x * y, [1, 2, 3, 4])  # returns: 24
functools.reduce(operator.mul, [1, 2, 3, 4])        # returns: 24
```

The `operator` module has functions for all of Python's operators specifically to make functional-style programming easier. Here, in the second line, we have used the `operator.mul()` function rather than having to create a multiplication function using `lambda` as we did in the first line.

Python also provides some built-in reducing functions: `all()`, which given an iterable, returns `True` if all the iterable's items return `True` when `bool()` is applied to them; `any()`, which returns `True` if any of the iterable's items is `True`; `max()`, which returns the largest item in the iterable; `min()`, which returns the smallest item in the iterable; and `sum()`, which returns the sum of the iterable's items.

Now that we have covered the key concepts, let us look at a few more examples. We will start with a couple of ways to get the total size of all the files in list `files`:

```
functools.reduce(operator.add, (os.path.getsize(x) for x in files))
functools.reduce(operator.add, map(os.path.getsize, files))
```

Using `map()` is often shorter than the equivalent list comprehension or generator expression except where there is a condition. We've used `operator.add()` as the addition function instead of `lambda x, y: x + y`.

If we only wanted to count the `.py` file sizes we can filter out non-Python files. Here are three ways to do this:

```
functools.reduce(operator.add, map(os.path.getsize,
            filter(lambda x: x.endswith(".py"), files)))
functools.reduce(operator.add, map(os.path.getsize,
            (x for x in files if x.endswith(".py"))))
functools.reduce(operator.add, (os.path.getsize(x)
            for x in files if x.endswith(".py")))
```

Arguably, the second and third versions are better because they don't require us to create a `lambda` function, but the choice between using generator expressions (or list comprehensions) and `map()` and `filter()` is most often purely a matter of personal programming style.

Using `map()`, `filter()`, and `functools.reduce()` often leads to the elimination of loops, as the examples we have seen illustrate. These functions are useful when converting code written in a functional language, but in Python we can usually replace `map()` with a list comprehension and `filter()` with a list comprehension with a condition, and many cases of `functools.reduce()` can be eliminated by using one of Python's built-in functional functions such as `all()`, `any()`, `max()`, `min()`, and `sum()`. For example:

```
sum(os.path.getsize(x) for x in files if x.endswith(".py"))
```

This achieves the same thing as the previous three examples, but is much more compact.

In addition to providing functions for Python's operators, the `operator` module also provides the `operator.attrgetter()` and `operator.itemgetter()` functions, the first of which we briefly met earlier in this short cut. Both of these return functions which can then be called to extract the specified attributes or items.

operator. attrgetter()

33

Whereas slicing can be used to extract a sequence of part of a list, and slicing with striding can be used to extract a sequence of parts (say, every third item with

`L[::3]`), `operator.itemgetter()` can be used to extract a sequence of arbitrary parts, for example, `operator.itemgetter(4, 5, 6, 11, 18)(L)`. The function returned by `operator.itemgetter()` does not have to be called immediately and thrown away as we have done here; it could be kept and passed as the function argument to `map()`, `filter()`, or `functools.reduce()`, or used in a dictionary, list, or set comprehension.

When we want to sort we can specify a key function. This function can be any function, for example, a `lambda` function, a built-in function or method (such as `str.lower()`), or a function returned by `operator.attrgetter()`. For example, assuming list `L` holds objects with a `priority` attribute, we can sort the list into priority order like this: `L.sort(key=operator.attrgetter("priority"))`.

In addition to the `functools` and `operator` modules already mentioned, the `iter-tools` module can also be useful for functional-style programming. For example, although it is possible to iterate over two or more lists by concatenating them, an alternative is to use `itertools.chain()` like this:

```
for value in itertools.chain(data_list1, data_list2, data_list3):
    total += value
```

The `itertools.chain()` function returns an iterator that gives successive values from the first sequence it is given, then successive values from the second sequence, and so on until all the values from all the sequences are used. The `itertools` module has many other functions and its documentation gives many small yet useful examples and is well worth reading.

16.1. Partial Function Application

Partial function application is the creation of a function from an existing function and some arguments to produce a new function that does what the original function did, but with some arguments fixed so that callers don't have to pass them. Here's a very simple example:

```
enumerate1 = functools.partial(enumerate, start=1)
for lino, line in enumerate1(lines):
    process_line(i, line)
```

The first line creates a new function, `enumerate1()`, that wraps the given function (`enumerate()`) and a keyword argument (`start=1`) so that when `enumerate1()` is called it calls the original function with the fixed argument—and with any other arguments that are given at the time it is called, in this case `lines`. Here we have used the `enumerate1()` function to provide conventional line counting starting from line 1.

Using partial function application can simplify our code, especially when we want to call the same functions with the same arguments again and again. For example, instead of specifying the mode and encoding arguments every time we call `open()` to process UTF-8 encoded text files, we could create a couple of functions with these arguments fixed:

```
reader = functools.partial(open, mode="rt", encoding="utf8")
writer = functools.partial(open, mode="wt", encoding="utf8")
```

Now we can open text files for reading by calling `reader(`*`filename`*`)` and for writing by calling `writer(`*`filename`*`)`.

One very common use case for partial function application is in GUI (Graphical User Interface) programming, where it is often convenient to have one particular function called when any one of a set of buttons is pressed. For example:

```
loadButton = tkinter.Button(frame, text="Load",
                command=functools.partial(doAction, "load"))
saveButton = tkinter.Button(frame, text="Save",
                command=functools.partial(doAction, "save"))
```

This example uses the `tkinter` GUI library that comes as standard with Python. The `tkinter.Button` class is used for buttons—here we have created two, both contained inside the same frame, and each with a text that indicates its purpose. Each button's `command` argument is set to the function that `tkinter` must call when the button is pressed, in this case the `doAction()` function. We have used partial function application to ensure that the first argument given to the `doAction()` function is a string that indicates which button called it so that `doAction()` is able to decide what action to perform.

Chapter 17. Descriptors with Class Decorators

In this section we combine descriptors with class decorators to create a powerful mechanism for creating validated attributes.

Descriptors 37☞

Class decorators 42☞

Up to now if we wanted to ensure that an attribute was set to only a valid value we have relied on properties (or used getter and setter methods). The disadvantage of such approaches is that we must add validating code for every attribute in every class that needs it. What would be much more convenient and easier to maintain, is if we could add attributes to classes with the necessary validation built in. Here is an example of the syntax we would like to use:

```
@valid_string("name", empty_allowed=False)
@valid_string("productid", empty_allowed=False,
          regex=re.compile(r"[A-Z]{3}\d{4}"))
@valid_string("category", empty_allowed=False, acceptable=
      frozenset(["Consumables", "Hardware", "Software", "Media"]))
@valid_number("price", minimum=0, maximum=1e6)
@valid_number("quantity", minimum=1, maximum=1000)
class StockItem:

    def __init__(self, name, productid, category, price, quantity):
        self.name = name
        self.productid = productid
        self.category = category
        self.price = price
        self.quantity = quantity
```

The `StockItem` class's attributes are all validated. For example, the `productid` attribute can be set only to a nonempty string that starts with three uppercase letters and ends with four digits, the `category` attribute can be set only to a nonempty string that is one of the specified values, and the `quantity` attribute can be set only to a number between 1 and 1 000 inclusive. If we try to set an invalid value an exception is raised.

The validation is achieved by combining class decorators with descriptors. As we noted earlier, class decorators can take only a single argument—the class they are to decorate. So here we have used the technique shown when we first discussed class decorators, and have the `valid_string()` and `valid_number()` functions take whatever arguments we want, and then return a decorator, which in turn takes the class and returns a modified version of the class.

Class decorators

42

Let's now look at the `valid_string()` function:

```
def valid_string(attr_name, empty_allowed=True, regex=None,
                 acceptable=None):
    def decorator(cls):

    name = "__" + attr_name
    def getter(self):
        return getattr(self, name)
    def setter(self, value):
        assert isinstance(value, str), (attr_name +
                                        " must be a string")
        if not empty_allowed and not value:
            raise ValueError("{0} may not be empty".format(
                             attr_name))
        if ((acceptable is not None and value not in acceptable) or
            (regex is not None and not regex.match(value))):
            raise ValueError("{0} cannot be set to {1}".format(
                             attr_name, value))
        setattr(self, name, value)
    setattr(cls, attr_name, GenericDescriptor(getter, setter))
    return cls
return decorator
```

The function starts by creating a class decorator function which takes a class as its sole argument. The decorator adds two attributes to the class it decorates: a private data attribute and a descriptor. For example, when the `valid_string()` function is called with the name "productid", the `StockItem` class gains the attribute `__productid` which holds the product ID's value, and the descriptor `productid` attribute which is used to access the value. For example, if we create an item using `item = StockItem("TV", "TVA4312", "Electrical", 500, 1)`, we can get the product ID using `item.productid` and set it using, for example, `item.productid = "TVB2100"`.

The getter function created by the decorator simply uses the global `getattr()` function to return the value of the private data attribute. The setter function incorporates the validation, and at the end, uses `setattr()` to set the private data attribute to the new (and valid) value. In fact, the private data attribute is only created the first time it is set.

Once the getter and setter functions have been created we use `setattr()` once again, this time to create a new class attribute with the given name (e.g., `productid`), and with its value set to be a descriptor of type `GenericDescriptor`. At the end, the decorator function returns the modified class, and the `valid_string()` function returns the decorator function.

The `valid_number()` function is structurally identical to the `valid_string()` function, only differing in the arguments it accepts and in the validation code in the setter, so we won't show it here. (The complete source code is in the `Valid.py` module.)

The last thing we need to cover is the `GenericDescriptor`, and that turns out to be the easiest part:

```
<a id="page_65/">class GenericDescriptor:

  def __init__(self, getter, setter):
    self.getter = getter
    self.setter = setter

  def __get__(self, instance, owner=None):
    if instance is None:
      return self
    return self.getter(instance)

  def __set__(self, instance, value):
    return self.setter(instance, value)
```

The descriptor is used to hold the getter and setter functions for each attribute and simply passes on the work of getting and setting to those functions.

Chapter 18. About the Author

Mark Summerfield is a computer science graduate with many years experience working in the software industry, primarily as a programmer. He also spent almost three years as Trolltech's documentation manager during which he founded and edited Trolltech's technical journal, *Qt Quarterly*. (Trolltech is now Nokia's Qt Software.) Mark is the coauthor of *C++ GUI Programming with Qt 4*, and author of *Rapid GUI Programming with Python and Qt: The Definitive Guide to PyQt Programming* and *Programming in Python 3: A Complete Introduction to the Python Language* (from which this short cut is taken). Mark owns Qtrac Ltd., www.qtrac.eu, where he works as an independent author, editor, trainer, and consultant, specializing in C++, Qt, Python, and PyQt.