CHAPTER 5
First-Class Functions

I have never considered Python to be heavily influenced by functional languages, no matter what people say or think. I was much more familiar with imperative languages such as C and Algol 68 and although I had made functions first-class objects, I didn't view Python as a functional programming language.[1]

— Guido van Rossum
Python BDFL

Functions in Python are first-class objects. Programming language theorists define a "first-class object" as a program entity that can be:

- Created at runtime
- Assigned to a variable or element in a data structure
- Passed as an argument to a function
- Returned as the result of a function

Integers, strings, and dictionaries are other examples of first-class objects in Python—nothing fancy here. But if you came to Python from a language where functions are not first-class citizens, this chapter and the rest of Part III of the book focuses on the implications and practical applications of treating functions as objects.

The term "first-class functions" is widely used as shorthand for "functions as first-class objects." It's not perfect because it seems to imply an "elite" among functions. In Python, all functions are first-class.

1. "Origins of Python's *Functional* Features", from Guido's The History of Python blog.

Treating a Function Like an Object

The console session in Example 5-1 shows that Python functions are objects. Here we create a function, call it, read its `__doc__` attribute, and check that the function object itself is an instance of the `function` class.

Example 5-1. Create and test a function, then read its `__doc__` and check its type

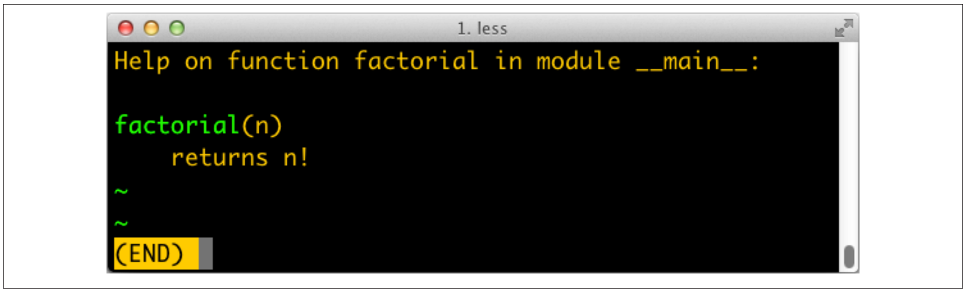
```
>>> def factorial(n):  ❶
...     '''returns n!'''
...     return 1 if n < 2 else n * factorial(n-1)
...
>>> factorial(42)
1405006117752879898543142606244511569936384000000000
>>> factorial.__doc__  ❷
'returns n!'
>>> type(factorial)  ❸
<class 'function'>
```

❶ This is a console session, so we're creating a function in "runtime."

❷ `__doc__` is one of several attributes of function objects.

❸ `factorial` is an instance of the `function` class.

The `__doc__` attribute is used to generate the help text of an object. In the Python interactive console, the command `help(factorial)` will display a screen like that in Figure 5-1.



```
Help on function factorial in module __main__:

factorial(n)
    returns n!
~
~
(END)
```

Figure 5-1. Help screen for the factorial function; the text is from the `__doc__` attribute of the function object

Example 5-2 shows the "first class" nature of a function object. We can assign it a variable `fact` and call it through that name. We can also pass `factorial` as an argument to `map`. The `map` function returns an iterable where each item is the result of the application of the first argument (a function) to succesive elements of the second argument (an iterable), `range(10)` in this example.

Example 5-2. Use function through a different name, and pass function as argument

```
>>> fact = factorial
>>> fact
<function factorial at 0x...>
>>> fact(5)
120
>>> map(factorial, range(11))
<map object at 0x...>
>>> list(map(fact, range(11)))
[1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800]
```

Having first-class functions enables programming in a functional style. One of the hallmarks of functional programming is the use of higher-order functions, our next topic.

Higher-Order Functions

A function that takes a function as argument or returns a function as the result is a *higher-order function*. One example is `map`, shown in Example 5-2. Another is the built-in function `sorted`: an optional `key` argument lets you provide a function to be applied to each item for sorting, as seen in "list.sort and the sorted Built-In Function" on page 42.

For example, to sort a list of words by length, simply pass the `len` function as the `key`, as in Example 5-3.

Example 5-3. Sorting a list of words by length

```
>>> fruits = ['strawberry', 'fig', 'apple', 'cherry', 'raspberry', 'banana']
>>> sorted(fruits, key=len)
['fig', 'apple', 'cherry', 'banana', 'raspberry', 'strawberry']
>>>
```

Any one-argument function can be used as the key. For example, to create a rhyme dictionary it might be useful to sort each word spelled backward. In Example 5-4, note that the words in the list are not changed at all; only their reversed spelling is used as the sort criterion, so that the berries appear together.

Example 5-4. Sorting a list of words by their reversed spelling

```
>>> def reverse(word):
...     return word[::-1]
>>> reverse('testing')
'gnitset'
>>> sorted(fruits, key=reverse)
['banana', 'apple', 'fig', 'raspberry', 'strawberry', 'cherry']
>>>
```

In the functional programming paradigm, some of the best known higher-order functions are `map`, `filter`, `reduce`, and `apply`. The `apply` function was deprecated in Python 2.3 and removed in Python 3 because it's no longer necessary. If you need to call a function with a dynamic set of arguments, you can just write `fn(*args, **keywords)` instead of `apply(fn, args, kwargs)`.

The `map`, `filter`, and `reduce` higher-order functions are still around, but better alternatives are available for most of their use cases, as the next section shows.

Modern Replacements for map, filter, and reduce

Functional languages commonly offer the `map`, `filter`, and `reduce` higher-order functions (sometimes with different names). The `map` and `filter` functions are still built-ins in Python 3, but since the introduction of list comprehensions and generator expressions, they are not as important. A listcomp or a genexp does the job of `map` and `filter` combined, but is more readable. Consider Example 5-5.

Example 5-5. Lists of factorials produced with map and filter compared to alternatives coded as list comprehensions

```
>>> list(map(fact, range(6)))  ❶
[1, 1, 2, 6, 24, 120]
>>> [fact(n) for n in range(6)]  ❷
[1, 1, 2, 6, 24, 120]
>>> list(map(factorial, filter(lambda n: n % 2, range(6))))  ❸
[1, 6, 120]
>>> [factorial(n) for n in range(6) if n % 2]  ❹
[1, 6, 120]
>>>
```

❶ Build a list of factorials from 0! to 5!.

❷ Same operation, with a list comprehension.

❸ List of factorials of odd numbers up to 5!, using both `map` and `filter`.

❹ List comprehension does the same job, replacing `map` and `filter`, and making `lambda` unnecessary.

In Python 3, `map` and `filter` return generators—a form of iterator—so their direct substitute is now a generator expression (in Python 2, these functions returned lists, therefore their closest alternative is a listcomp).

The `reduce` function was demoted from a built-in in Python 2 to the `functools` module in Python 3. Its most common use case, summation, is better served by the `sum` built-in available since Python 2.3 was released in 2003. This is a big win in terms of readability and performance (see Example 5-6).

Example 5-6. Sum of integers up to 99 performed with reduce and sum

```
>>> from functools import reduce  ❶
>>> from operator import add  ❷
>>> reduce(add, range(100))  ❸
```

```
4950
>>> sum(range(100))  ❹
4950
>>>
```

❶ Starting with Python 3.0, `reduce` is not a built-in.

❷ Import `add` to avoid creating a function just to add two numbers.

❸ Sum integers up to 99.

❹ Same task using `sum`; import or adding function not needed.

The common idea of `sum` and `reduce` is to apply some operation to successive items in a sequence, accumulating previous results, thus reducing a sequence of values to a single value.

Other reducing built-ins are `all` and `any`:

`all(iterable)`
: Returns `True` if every element of the `iterable` is truthy; `all([])` returns `True`.

`any(iterable)`
: Returns `True` if any element of the `iterable` is truthy; `any([])` returns `False`.

I give a fuller explanation of `reduce` in "Vector Take #4: Hashing and a Faster ==" on page 288 where an ongoing example provides a meaningful context for the use of this function. The reducing functions are summarized later in the book when iterables are in focus, in "Iterable Reducing Functions" on page 434.

To use a higher-order function, sometimes it is convenient to create a small, one-off function. That is why anonymous functions exist. We'll cover them next.

Anonymous Functions

The `lambda` keyword creates an anonymous function within a Python expression.

However, the simple syntax of Python limits the body of lambda functions to be pure expressions. In other words, the body of a `lambda` cannot make assignments or use any other Python statement such as `while`, `try`, etc.

The best use of anonymous functions is in the context of an argument list. For example, Example 5-7 is the rhyme index example from Example 5-4 rewritten with `lambda`, without defining a `reverse` function.

Example 5-7. Sorting a list of words by their reversed spelling using lambda

```
>>> fruits = ['strawberry', 'fig', 'apple', 'cherry', 'raspberry', 'banana']
>>> sorted(fruits, key=lambda word: word[::-1])
```

```
['banana', 'apple', 'fig', 'raspberry', 'strawberry', 'cherry']
>>>
```

Outside the limited context of arguments to higher-order functions, anonymous functions are rarely useful in Python. The syntactic restrictions tend to make nontrivial lambdas either unreadable or unworkable.

Lundh's lambda Refactoring Recipe

If you find a piece of code hard to understand because of a `lambda`, Fredrik Lundh suggests this refactoring procedure:

1. Write a comment explaining what the heck that `lambda` does.
2. Study the comment for a while, and think of a name that captures the essence of the comment.
3. Convert the `lambda` to a `def` statement, using that name.
4. Remove the comment.

These steps are quoted from the Functional Programming HOWTO, a must read.

The `lambda` syntax is just syntactic sugar: a `lambda` expression creates a function object just like the `def` statement. That is just one of several kinds of callable objects in Python. The following section reviews all of them.

The Seven Flavors of Callable Objects

The call operator (i.e., `()`) may be applied to other objects beyond user-defined functions. To determine whether an object is callable, use the `callable()` built-in function. The Python Data Model documentation lists seven callable types:

User-defined functions
: Created with `def` statements or `lambda` expressions.

Built-in functions
: A function implemented in C (for CPython), like `len` or `time.strftime`.

Built-in methods
: Methods implemented in C, like `dict.get`.

Methods
: Functions defined in the body of a class.

Classes
: When invoked, a class runs its `__new__` method to create an instance, then `__init__` to initialize it, and finally the instance is returned to the caller. Because there is no `new` operator in Python, calling a class is like calling a function. (Usually calling a class creates an instance of the same class, but other behaviors are possible by overriding `__new__`. We'll see an example of this in "Flexible Object Creation with `__new__`" on page 592.)

Class instances
: If a class defines a `__call__` method, then its instances may be invoked as functions. See "User-Defined Callable Types" on page 145.

Generator functions
: Functions or methods that use the `yield` keyword. When called, generator functions return a generator object.

Generator functions are unlike other callables in many respects. Chapter 14 is devoted to them. They can also be used as coroutines, which are covered in Chapter 16.

Given the variety of existing callable types in Python, the safest way to determine whether an object is callable is to use the `callable()` built-in:

```
>>> abs, str, 13
(<built-in function abs>, <class 'str'>, 13)
>>> [callable(obj) for obj in (abs, str, 13)]
[True, True, False]
```

We now move on to building class instances that work as callable objects.

User-Defined Callable Types

Not only are Python functions real objects, but arbitrary Python objects may also be made to behave like functions. Implementing a `__call__` instance method is all it takes.

Example 5-8 implements a `BingoCage` class. An instance is built from any iterable, and stores an internal `list` of items, in random order. Calling the instance pops an item.

Example 5-8. bingocall.py: A BingoCage does one thing: picks items from a shuffled list

```
import random

class BingoCage:

    def __init__(self, items):
        self._items = list(items)  ❶
        random.shuffle(self._items)  ❷
    def pick(self):  ❸
        try:
            return self._items.pop()
        except IndexError:
            raise LookupError('pick from empty BingoCage')  ❹

    def __call__(self):  ❺
        return self.pick()
```

❶ `__init__` accepts any iterable; building a local copy prevents unexpected side effects on any `list` passed as an argument.

❷ `shuffle` is guaranteed to work because `self._items` is a `list`.

❸ The main method.

❹ Raise exception with custom message if `self._items` is empty.

❺ Shortcut to `bingo.pick()`: `bingo()`.

Here is a simple demo of Example 5-8. Note how a `bingo` instance can be invoked as a function, and the `callable(…)` built-in recognizes it as a callable object:

```
>>> bingo = BingoCage(range(3))
>>> bingo.pick()
1
>>> bingo()
0
>>> callable(bingo)
True
```

A class implementing `__call__` is an easy way to create function-like objects that have some internal state that must be kept across invocations, like the remaining items in the `BingoCage`. An example is a decorator. Decorators must be functions, but it is sometimes convenient to be able to "remember" something between calls of the decorator (e.g., for memoization—caching the results of expensive computations for later use).

A totally different approach to creating functions with internal state is to use closures. Closures, as well as decorators, are the subject of Chapter 7.

We now move on to another aspect of handling functions as objects: runtime introspection.

Function Introspection

Function objects have many attributes beyond `__doc__`. See what the `dir` function reveals about our `factorial`:

```
>>> dir(factorial)
['__annotations__', '__call__', '__class__', '__closure__', '__code__',
'__defaults__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__get__', '__getattribute__', '__globals__',
'__gt__', '__hash__', '__init__', '__kwdefaults__', '__le__', '__lt__',
'__module__', '__name__', '__ne__', '__new__', '__qualname__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__']
>>>
```

Most of these attributes are common to Python objects in general. In this section, we cover those that are especially relevant to treating functions as objects, starting with `__dict__`.

Like the instances of a plain user-defined class, a function uses the `__dict__` attribute to store user attributes assigned to it. This is useful as a primitive form of annotation. Assigning arbitrary attributes to functions is not a very common practice in general, but Django is one framework that uses it. See, for example, the `short_description`, `boolean`, and `allow_tags` attributes described in The Django admin site documentation. In the Django docs, this example shows attaching a `short_description` to a method, to determine the description that will appear in record listings in the Django admin when that method is used:

```
def upper_case_name(obj):
    return ("%s %s" % (obj.first_name, obj.last_name)).upper()
upper_case_name.short_description = 'Customer name'
```

Now let us focus on the attributes that are specific to functions and are not found in a generic Python user-defined object. Computing the difference of two sets quickly gives us a list of the function-specific attributes (see Example 5-9).

Example 5-9. Listing attributes of functions that don't exist in plain instances

```
>>> class C: pass  # ❶
>>> obj = C()  # ❷
>>> def func(): pass  # ❸
>>> sorted(set(dir(func)) - set(dir(obj))) # ❹
['__annotations__', '__call__', '__closure__', '__code__', '__defaults__',
'__get__', '__globals__', '__kwdefaults__', '__name__', '__qualname__']
>>>
```

❶ Create bare user-defined class.

❷ Make an instance of it.

❸ Create a bare function.

❹ Using set difference, generate a sorted list of the attributes that exist in a function but not in an instance of a bare class.

Table 5-1 shows a summary of the attributes listed by Example 5-9.

Table 5-1. Attributes of user-defined functions

Name	Type	Description
`__annotations__`	`dict`	Parameter and return annotations
`__call__`	`method-wrapper`	Implementation of the () operator; a.k.a. the callable object protocol
`__closure__`	`tuple`	The function closure, i.e., bindings for free variables (often is None)
`__code__`	`code`	Function metadata and function body compiled into bytecode
`__defaults__`	`tuple`	Default values for the formal parameters
`__get__`	`method-wrapper`	Implementation of the read-only descriptor protocol (see Chapter 20)
`__globals__`	`dict`	Global variables of the module where the function is defined
`__kwdefaults__`	`dict`	Default values for the keyword-only formal parameters
`__name__`	`str`	The function name
`__qualname__`	`str`	The qualified function name, e.g., `Random.choice` (see PEP-3155)

We will discuss the `__defaults__`, `__code__`, and `__annotations__` functions, used by IDEs and frameworks to extract information about function signatures, in later sections. But to fully appreciate these attributes, we will make a detour to explore the powerful syntax Python offers to declare function parameters and to pass arguments into them.

From Positional to Keyword-Only Parameters

One of the best features of Python functions is the extremely flexible parameter handling mechanism, enhanced with keyword-only arguments in Python 3. Closely related are the use of `*` and `**` to "explode" iterables and mappings into separate arguments when we call a function. To see these features in action, see the code for Example 5-10 and tests showing its use in Example 5-11.

Example 5-10. tag generates HTML; a keyword-only argument cls is used to pass "class" attributes as a workaround because class is a keyword in Python

```
def tag(name, *content, cls=None, **attrs):
    """Generate one or more HTML tags"""
    if cls is not None:
        attrs['class'] = cls
    if attrs:
        attr_str = ''.join(' %s="%s"' % (attr, value)
                           for attr, value
                           in sorted(attrs.items()))
    else:
        attr_str = ''
    if content:
        return '\n'.join('<%s%s>%s</%s>' %
                         (name, attr_str, c, name) for c in content)
    else:
        return '<%s%s />' % (name, attr_str)
```

The `tag` function can be invoked in many ways, as Example 5-11 shows.

Example 5-11. Some of the many ways of calling the tag function from Example 5-10

```
>>> tag('br')  ❶
'<br />'
>>> tag('p', 'hello')  ❷
'<p>hello</p>'
>>> print(tag('p', 'hello', 'world'))
<p>hello</p>
<p>world</p>
>>> tag('p', 'hello', id=33)  ❸
'<p id="33">hello</p>'
>>> print(tag('p', 'hello', 'world', cls='sidebar'))  ❹
<p class="sidebar">hello</p>
<p class="sidebar">world</p>
>>> tag(content='testing', name="img")  ❺
'<img content="testing" />'
>>> my_tag = {'name': 'img', 'title': 'Sunset Boulevard',
...           'src': 'sunset.jpg', 'cls': 'framed'}
>>> tag(**my_tag)  ❻
'<img class="framed" src="sunset.jpg" title="Sunset Boulevard" />'
```

❶ A single positional argument produces an empty tag with that name.

❷ Any number of arguments after the first are captured by `*content` as a `tuple`.

❸ Keyword arguments not explicitly named in the `tag` signature are captured by `**attrs` as a `dict`.

❹ The `cls` parameter can only be passed as a keyword argument.

❺ Even the first positional argument can be passed as a keyword when `tag` is called.

❻ Prefixing the `my_tag` `dict` with `**` passes all its items as separate arguments, which are then bound to the named parameters, with the remaining caught by `**attrs`.

Keyword-only arguments are a new feature in Python 3. In Example 5-10, the `cls` parameter can only be given as a keyword argument—it will never capture unnamed positional arguments. To specify keyword-only arguments when defining a function, name them after the argument prefixed with `*`. If you don't want to support variable positional arguments but still want keyword-only arguments, put a `*` by itself in the signature, like this:

```
>>> def f(a, *, b):
...     return a, b
...
>>> f(1, b=2)
(1, 2)
```

Note that keyword-only arguments do not need to have a default value: they can be mandatory, like b in the preceding example.

We now move on to the introspection of function parameters, starting with a motivating example from a web framework, and on through introspection techniques.

Retrieving Information About Parameters

An interesting application of function introspection can be found in the Bobo HTTP micro-framework. To see that in action, consider a variation of the Bobo tutorial "Hello world" application in Example 5-12.

Example 5-12. Bobo knows that hello requires a person argument, and retrieves it from the HTTP request

```
import bobo

@bobo.query('/')
def hello(person):
    return 'Hello %s!' % person
```

The `bobo.query` decorator integrates a plain function such as `hello` with the request handling machinery of the framework. We'll cover decorators in Chapter 7—that's not the point of this example here. The point is that Bobo introspects the `hello` function and finds out it needs one parameter named `person` to work, and it will retrieve a parameter with that name from the request and pass it to `hello`, so the programmer does not need to touch the request object at all.

If you install Bobo and point its development server to the script in Example 5-12 (e.g., `bobo -f hello.py`), a hit on the URL `http://localhost:8080/` will produce the message "Missing form variable person" with a 403 HTTP code. This happens because Bobo understands that the `person` argument is required to call `hello`, but no such name was found in the request. Example 5-13 is a shell session using `curl` to show this behavior.

Example 5-13. Bobo issues a 403 forbidden response if there are missing function arguments in the request; curl -i is used to dump the headers to standard output

```
$ curl -i http://localhost:8080/
HTTP/1.0 403 Forbidden
Date: Thu, 21 Aug 2014 21:39:44 GMT
Server: WSGIServer/0.2 CPython/3.4.1
Content-Type: text/html; charset=UTF-8
Content-Length: 103

<html>
<head><title>Missing parameter</title></head>
<body>Missing form variable person</body>
</html>
```

However, if you get http://localhost:8080/?person=Jim, the response will be the string 'Hello Jim!'. See Example 5-14.

Example 5-14. Passing the person parameter is required for an OK response

```
$ curl -i http://localhost:8080/?person=Jim
HTTP/1.0 200 OK
Date: Thu, 21 Aug 2014 21:42:32 GMT
Server: WSGIServer/0.2 CPython/3.4.1
Content-Type: text/html; charset=UTF-8
Content-Length: 10

Hello Jim!
```

How does Bobo know which parameter names are required by the function, and whether they have default values or not?

Within a function object, the `__defaults__` attribute holds a tuple with the default values of positional and keyword arguments. The defaults for keyword-only arguments appear in `__kwdefaults__`. The names of the arguments, however, are found within the `__code__` attribute, which is a reference to a `code` object with many attributes of its own.

To demonstrate the use of these attributes, we will inspect the function `clip` in a module *clip.py*, listed in Example 5-15.

Example 5-15. Function to shorten a string by clipping at a space near the desired length

```
def clip(text, max_len=80):
    """Return text clipped at the last space before or after max_len
    """
    end = None
    if len(text) > max_len:
        space_before = text.rfind(' ', 0, max_len)
        if space_before >= 0:
            end = space_before
        else:
            space_after = text.rfind(' ', max_len)
            if space_after >= 0:
                end = space_after
    if end is None:  # no spaces were found
        end = len(text)
    return text[:end].rstrip()
```

Example 5-16 shows the values of `__defaults__`, `__code__.co_varnames`, and `__code__.co_argcount` for the `clip` function listed in Example 5-15.

Example 5-16. Extracting information about the function arguments

```
>>> from clip import clip
>>> clip.__defaults__
```

```
(80,)
>>> clip.__code__  # doctest: +ELLIPSIS
<code object clip at 0x...>
>>> clip.__code__.co_varnames
('text', 'max_len', 'end', 'space_before', 'space_after')
>>> clip.__code__.co_argcount
2
```

As you can see, this is not the most convenient arrangement of information. The argument names appear in `__code__.co_varnames`, but that also includes the names of the local variables created in the body of the function. Therefore, the argument names are the first N strings, where N is given by `__code__.co_argcount` which—by the way—does not include any variable arguments prefixed with `*` or `**`. The default values are identified only by their position in the `__defaults__` tuple, so to link each with the respective argument, you have to scan from last to first. In the example, we have two arguments, `text` and `max_len`, and one default, `80`, so it must belong to the last argument, `max_len`. This is awkward.

Fortunately, there is a better way: the `inspect` module.

Take a look at Example 5-17.

Example 5-17. Extracting the function signature

```
>>> from clip import clip
>>> from inspect import signature
>>> sig = signature(clip)
>>> sig  # doctest: +ELLIPSIS
<inspect.Signature object at 0x...>
>>> str(sig)
'(text, max_len=80)'
>>> for name, param in sig.parameters.items():
...     print(param.kind, ':', name, '=', param.default)
...
POSITIONAL_OR_KEYWORD : text = <class 'inspect._empty'>
POSITIONAL_OR_KEYWORD : max_len = 80
```

This is much better. `inspect.signature` returns an `inspect.Signature` object, which has a `parameters` attribute that lets you read an ordered mapping of names to `inspect.Parameter` objects. Each `Parameter` instance has attributes such as `name`, `default`, and `kind`. The special value `inspect._empty` denotes parameters with no default, which makes sense considering that `None` is a valid—and popular—default value.

The `kind` attribute holds one of five possible values from the `_ParameterKind` class:

`POSITIONAL_OR_KEYWORD`
: A parameter that may be passed as a positional or as a keyword argument (most Python function parameters are of this kind).

VAR_POSITIONAL
: A tuple of positional parameters.

VAR_KEYWORD
: A dict of keyword parameters.

KEYWORD_ONLY
: A keyword-only parameter (new in Python 3).

POSITIONAL_ONLY
: A positional-only parameter; currently unsupported by Python function declaration syntax, but exemplified by existing functions implemented in C—like divmod—that do not accept parameters passed by keyword.

Besides name, default, and kind, inspect.Parameter objects have an annotation attribute that is usually inspect._empty but may contain function signature metadata provided via the new annotations syntax in Python 3 (annotations are covered in the next section).

An inspect.Signature object has a bind method that takes any number of arguments and binds them to the parameters in the signature, applying the usual rules for matching actual arguments to formal parameters. This can be used by a framework to validate arguments prior to the actual function invocation. Example 5-18 shows how.

Example 5-18. Binding the function signature from the tag function in Example 5-10 to a dict of arguments

```
>>> import inspect
>>> sig = inspect.signature(tag)  ❶
>>> my_tag = {'name': 'img', 'title': 'Sunset Boulevard',
...           'src': 'sunset.jpg', 'cls': 'framed'}
>>> bound_args = sig.bind(**my_tag)  ❷
>>> bound_args
<inspect.BoundArguments object at 0x...>  ❸
>>> for name, value in bound_args.arguments.items():  ❹
...     print(name, '=', value)
...
name = img
cls = framed
attrs = {'title': 'Sunset Boulevard', 'src': 'sunset.jpg'}
>>> del my_tag['name']  ❺
>>> bound_args = sig.bind(**my_tag)  ❻
Traceback (most recent call last):
  ...
TypeError: 'name' parameter lacking default value
```

❶ Get the signature from tag function in Example 5-10.

❷ Pass a dict of arguments to .bind().

❸ An `inspect.BoundArguments` object is produced.

❹ Iterate over the items in `bound_args.arguments`, which is an `OrderedDict`, to display the names and values of the arguments.

❺ Remove the mandatory argument `name` from `my_tag`.

❻ Calling `sig.bind(**my_tag)` raises a `TypeError` complaining of the missing `name` parameter.

This example shows how the Python data model, with the help of `inspect`, exposes the same machinery the interpreter uses to bind arguments to formal parameters in function calls.

Frameworks and tools like IDEs can use this information to validate code. Another feature of Python 3, function annotations, enhances the possible uses of this, as we will see next.

Function Annotations

Python 3 provides syntax to attach metadata to the parameters of a function declaration and its return value. Example 5-19 is an annotated version of Example 5-15. The only differences are in the first line.

Example 5-19. Annotated clip function

```
def clip(text:str, max_len:'int > 0'=80) -> str:  ❶
    """Return text clipped at the last space before or after max_len
    """
    end = None
    if len(text) > max_len:
        space_before = text.rfind(' ', 0, max_len)
        if space_before >= 0:
            end = space_before
        else:
            space_after = text.rfind(' ', max_len)
            if space_after >= 0:
                end = space_after
    if end is None:  # no spaces were found
        end = len(text)
    return text[:end].rstrip()
```

❶ The annotated function declaration.

Each argument in the function declaration may have an annotation expression preceded by `:`. If there is a default value, the annotation goes between the argument name and the `=` sign. To annotate the return value, add `->` and another expression between the `)` and the `:` at the tail of the function declaration. The expressions may be of any type. The most common types used in annotations are classes, like `str` or `int`, or strings, like `'int > 0'`, as seen in the annotation for `max_len` in Example 5-19.

No processing is done with the annotations. They are merely stored in the `__annotations__` attribute of the function, a `dict`:

```
>>> from clip_annot import clip
>>> clip.__annotations__
{'text': <class 'str'>, 'max_len': 'int > 0', 'return': <class 'str'>}
```

The item with key `'return'` holds the return value annotation marked with `->` in the function declaration in Example 5-19.

The only thing Python does with annotations is to store them in the `__annotations__` attribute of the function. Nothing else: no checks, enforcement, validation, or any other action is performed. In other words, annotations have no meaning to the Python interpreter. They are just metadata that may be used by tools, such as IDEs, frameworks, and decorators. At this writing no tools that use this metadata exist in the standard library, except that `inspect.signature()` knows how to extract the annotations, as Example 5-20 shows.

Example 5-20. Extracting annotations from the function signature

```
>>> from clip_annot import clip
>>> from inspect import signature
>>> sig = signature(clip)
>>> sig.return_annotation
<class 'str'>
>>> for param in sig.parameters.values():
...     note = repr(param.annotation).ljust(13)
...     print(note, ':', param.name, '=', param.default)
<class 'str'> : text = <class 'inspect._empty'>
'int > 0'     : max_len = 80
```

The `signature` function returns a `Signature` object, which has a `return_annotation` attribute and a `parameters` dictionary mapping parameter names to `Parameter` objects. Each `Parameter` object has its own `annotation` attribute. That's how Example 5-20 works.

In the future, frameworks such as Bobo could support annotations to further automate request processing. For example, an argument annotated as `price:float` may be automatically converted from a query string to the `float` expected by the function; a string annotation like `quantity:'int > 0'` might be parsed to perform conversion and validation of a parameter.

The biggest impact of function annotations will probably not be dynamic settings such as Bobo, but in providing optional type information for static type checking in tools like IDEs and linters.

After this deep dive into the anatomy of functions, the remainder of this chapter covers the most useful packages in the standard library that support functional programming.

Packages for Functional Programming

Although Guido makes it clear that Python does not aim to be a functional programming language, a functional coding style can be used to good extent, thanks to the support of packages like `operator` and `functools`, which we cover in the next two sections.

The operator Module

Often in functional programming it is convenient to use an arithmetic operator as a function. For example, suppose you want to multiply a sequence of numbers to calculate factorials without using recursion. To perform summation, you can use `sum`, but there is no equivalent function for multiplication. You could use `reduce`—as we saw in "Modern Replacements for map, filter, and reduce" on page 142—but this requires a function to multiply two items of the sequence. Example 5-21 shows how to solve this using `lambda`.

Example 5-21. Factorial implemented with reduce and an anonymous function

```
from functools import reduce

def fact(n):
    return reduce(lambda a, b: a*b, range(1, n+1))
```

To save you the trouble of writing trivial anonymous functions like `lambda a, b: a*b`, the `operator` module provides function equivalents for dozens of arithmetic operators. With it, we can rewrite Example 5-21 as Example 5-22.

Example 5-22. Factorial implemented with reduce and operator.mul

```
from functools import reduce
from operator import mul

def fact(n):
    return reduce(mul, range(1, n+1))
```

Another group of one-trick lambdas that `operator` replaces are functions to pick items from sequences or read attributes from objects: `itemgetter` and `attrgetter` actually build custom functions to do that.

Example 5-23 shows a common use of `itemgetter`: sorting a list of tuples by the value of one field. In the example, the cities are printed sorted by country code (field 1). Essentially, `itemgetter(1)` does the same as `lambda fields: fields[1]`: create a function that, given a collection, returns the item at index 1.

Example 5-23. Demo of itemgetter to sort a list of tuples (data from Example 2-8)

```
>>> metro_data = [
...     ('Tokyo', 'JP', 36.933, (35.689722, 139.691667)),
...     ('Delhi NCR', 'IN', 21.935, (28.613889, 77.208889)),
...     ('Mexico City', 'MX', 20.142, (19.433333, -99.133333)),
...     ('New York-Newark', 'US', 20.104, (40.808611, -74.020386)),
...     ('Sao Paulo', 'BR', 19.649, (-23.547778, -46.635833)),
... ]
>>>
>>> from operator import itemgetter
>>> for city in sorted(metro_data, key=itemgetter(1)):
...     print(city)
...
('Sao Paulo', 'BR', 19.649, (-23.547778, -46.635833))
('Delhi NCR', 'IN', 21.935, (28.613889, 77.208889))
('Tokyo', 'JP', 36.933, (35.689722, 139.691667))
('Mexico City', 'MX', 20.142, (19.433333, -99.133333))
('New York-Newark', 'US', 20.104, (40.808611, -74.020386))
```

If you pass multiple index arguments to `itemgetter`, the function it builds will return tuples with the extracted values:

```
    >>> cc_name = itemgetter(1, 0)
    >>> for city in metro_data:
    ...     print(cc_name(city))
    ...
    ('JP', 'Tokyo')
    ('IN', 'Delhi NCR')
    ('MX', 'Mexico City')
    ('US', 'New York-Newark')
    ('BR', 'Sao Paulo')
    >>>
```

Because `itemgetter` uses the `[]` operator, it supports not only sequences but also mappings and any class that implements `__getitem__`.

A sibling of `itemgetter` is `attrgetter`, which creates functions to extract object attributes by name. If you pass `attrgetter` several attribute names as arguments, it also returns a tuple of values. In addition, if any argument name contains a `.` (dot), `attrgetter` navigates through nested objects to retrieve the attribute. These behaviors are shown in Example 5-24. This is not the shortest console session because we need to build a nested structure to showcase the handling of dotted attributes by `attrgetter`.

Example 5-24. Demo of attrgetter to process a previously defined list of namedtuple called metro_data (the same list that appears in Example 5-23)

```
>>> from collections import namedtuple
>>> LatLong = namedtuple('LatLong', 'lat long')  # ❶
>>> Metropolis = namedtuple('Metropolis', 'name cc pop coord')  # ❷
>>> metro_areas = [Metropolis(name, cc, pop, LatLong(lat, long))  # ❸
...     for name, cc, pop, (lat, long) in metro_data]
>>> metro_areas[0]
Metropolis(name='Tokyo', cc='JP', pop=36.933, coord=LatLong(lat=35.689722,
long=139.691667))
>>> metro_areas[0].coord.lat  # ❹
35.689722
>>> from operator import attrgetter
>>> name_lat = attrgetter('name', 'coord.lat')  # ❺
>>>
>>> for city in sorted(metro_areas, key=attrgetter('coord.lat')):  # ❻
...     print(name_lat(city))  # ❼
...
('Sao Paulo', -23.547778)
('Mexico City', 19.433333)
('Delhi NCR', 28.613889)
('Tokyo', 35.689722)
('New York-Newark', 40.808611)
```

❶ Use `namedtuple` to define `LatLong`.

❷ Also define `Metropolis`.

❸ Build `metro_areas` list with `Metropolis` instances; note the nested tuple unpacking to extract `(lat, long)` and use them to build the `LatLong` for the `coord` attribute of `Metropolis`.

❹ Reach into element `metro_areas[0]` to get its latitude.

❺ Define an `attrgetter` to retrieve the `name` and the `coord.lat` nested attribute.

❻ Use `attrgetter` again to sort list of cities by latitude.

❼ Use the `attrgetter` defined in ❺ to show only city name and latitude.

Here is a partial list of functions defined in `operator` (names starting with `_` are omitted, because they are mostly implementation details):

```
>>> [name for name in dir(operator) if not name.startswith('_')]
['abs', 'add', 'and_', 'attrgetter', 'concat', 'contains',
'countOf', 'delitem', 'eq', 'floordiv', 'ge', 'getitem', 'gt',
'iadd', 'iand', 'iconcat', 'ifloordiv', 'ilshift', 'imod', 'imul',
'index', 'indexOf', 'inv', 'invert', 'ior', 'ipow', 'irshift',
'is_', 'is_not', 'isub', 'itemgetter', 'itruediv', 'ixor', 'le',
'length_hint', 'lshift', 'lt', 'methodcaller', 'mod', 'mul', 'ne',
'neg', 'not_', 'or_', 'pos', 'pow', 'rshift', 'setitem', 'sub',
'truediv', 'truth', 'xor']
```

Most of the 52 names listed are self-evident. The group of names prefixed with `i` and the name of another operator—e.g., `iadd`, `iand`, etc.—correspond to the augmented assignment operators—e.g., `+=`, `&=`, etc. These change their first argument in place, if it is mutable; if not, the function works like the one without the `i` prefix: it simply returns the result of the operation.

Of the remaining `operator` functions, `methodcaller` is the last we will cover. It is somewhat similar to `attrgetter` and `itemgetter` in that it creates a function on the fly. The function it creates calls a method by name on the object given as argument, as shown in Example 5-25.

Example 5-25. Demo of methodcaller: second test shows the binding of extra arguments

```
>>> from operator import methodcaller
>>> s = 'The time has come'
>>> upcase = methodcaller('upper')
>>> upcase(s)
'THE TIME HAS COME'
>>> hiphenate = methodcaller('replace', ' ', '-')
>>> hiphenate(s)
'The-time-has-come'
```

The first test in Example 5-25 is there just to show `methodcaller` at work, but if you need to use the `str.upper` as a function, you can just call it on the `str` class and pass a string as argument, like this:

```
    >>> str.upper(s)
    'THE TIME HAS COME'
```

The second test in Example 5-25 shows that `methodcaller` can also do a partial application to freeze some arguments, like the `functools.partial` function does. That is our next subject.

Freezing Arguments with functools.partial

The `functools` module brings together a handful of higher-order functions. The best known of them is probably `reduce`, which was covered in "Modern Replacements for map, filter, and reduce" on page 142. Of the remaining functions in `functools`, the most useful is `partial` and its variation, `partialmethod`.

`functools.partial` is a higher-order function that allows partial application of a function. Given a function, a partial application produces a new callable with some of the arguments of the original function fixed. This is useful to adapt a function that takes one or more arguments to an API that requires a callback with fewer arguments. Example 5-26 is a trivial demonstration.

Example 5-26. Using partial to use a two-argument function where a one-argument callable is required

```
>>> from operator import mul
>>> from functools import partial
>>> triple = partial(mul, 3)  ❶
>>> triple(7)  ❷
21
>>> list(map(triple, range(1, 10)))  ❸
[3, 6, 9, 12, 15, 18, 21, 24, 27]
```

❶ Create new `triple` function from `mul`, binding first positional argument to `3`.

❷ Test it.

❸ Use `triple` with `map`; `mul` would not work with `map` in this example.

A more useful example involves the `unicode.normalize` function that we saw in "Normalizing Unicode for Saner Comparisons" on page 117. If you work with text from many languages, you may want to apply `unicode.normalize('NFC', s)` to any string `s` before comparing or storing it. If you do that often, it's handy to have an `nfc` function to do so, as in Example 5-27.

Example 5-27. Building a convenient Unicode normalizing function with partial

```
>>> import unicodedata, functools
>>> nfc = functools.partial(unicodedata.normalize, 'NFC')
>>> s1 = 'café'
>>> s2 = 'cafe\u0301'
>>> s1, s2
('café', 'café')
>>> s1 == s2
False
>>> nfc(s1) == nfc(s2)
True
```

`partial` takes a callable as first argument, followed by an arbitrary number of positional and keyword arguments to bind.

Example 5-28 shows the use of `partial` with the `tag` function from Example 5-10, to freeze one positional argument and one keyword argument.

Example 5-28. Demo of partial applied to the function tag from Example 5-10

```
>>> from tagger import tag
>>> tag
<function tag at 0x10206d1e0>  ❶
>>> from functools import partial
>>> picture = partial(tag, 'img', cls='pic-frame')  ❷
>>> picture(src='wumpus.jpeg')
'<img class="pic-frame" src="wumpus.jpeg" />'  ❸
>>> picture
functools.partial(<function tag at 0x10206d1e0>, 'img', cls='pic-frame')  ❹
>>> picture.func  ❺
<function tag at 0x10206d1e0>
>>> picture.args
('img',)
```

```
>>> picture.keywords
{'cls': 'pic-frame'}
```

❶ Import `tag` from Example 5-10 and show its ID.

❷ Create `picture` function from `tag` by fixing the first positional argument with `'img'` and the `cls` keyword argument with `'pic-frame'`.

❸ `picture` works as expected.

❹ `partial()` returns a `functools.partial` object.[2]

❺ A `functools.partial` object has attributes providing access to the original function and the fixed arguments.

The `functools.partialmethod` function (new in Python 3.4) does the same job as `partial`, but is designed to work with methods.

An impressive `functools` function is `lru_cache`, which does memoization—a form of automatic optimization that works by storing the results of function calls to avoid expensive recalculations. We will cover it in Chapter 7, where decorators are explained, along with other higher-order functions designed to be used as decorators: `singledispatch` and `wraps`.

Chapter Summary

The goal of this chapter was to explore the first-class nature of functions in Python. The main ideas are that you can assign functions to variables, pass them to other functions, store them in data structures, and access function attributes, allowing frameworks and tools to act on that information. Higher-order functions, a staple of functional programming, are common in Python—even if the use of `map`, `filter`, and `reduce` is not as frequent as it was—thanks to list comprehensions (and similar constructs like generator expressions) and the appearance of reducing built-ins like `sum`, `all`, and `any`. The `sorted`, `min`, `max` built-ins, and `functools.partial` are examples of commonly used higher-order functions in the language.

Callables come in seven different flavors in Python, from the simple functions created with `lambda` to instances of classes implementing `__call__`. They can all be detected by the `callable()` built-in. Every callable supports the same rich syntax for declaring formal parameters, including keyword-only parameters and annotations—both new features introduced with Python 3.

2. The source code for `functools.py` reveals that the `functools.partial` class is implemented in C and is used by default. If that is not available, a pure-Python implementation of `partial` is available since Python 3.4.in the `functools` module.

Python functions and their annotations have a rich set of attributes that can be read with the help of the `inspect` module, which includes the `Signature.bind` method to apply the flexible rules that Python uses to bind actual arguments to declared parameters.

Lastly, we covered some functions from the `operator` module and `functools.partial`, which facilitate functional programming by minimizing the need for the functionally challenged `lambda` syntax.

Further Reading

The next two chapters continue our exploration of programming with function objects. Chapter 6 shows how first-class functions can simplify some classic object-oriented design patterns, while Chapter 7 dives into function decorators—a special kind of higher-order function—and the closure mechanism that makes them work.

Chapter 7 of the *Python Cookbook, Third Edition* (O'Reilly), by David Beazley and Brian K. Jones, is an excellent complement to the current chapter as well as Chapter 7 of this book, covering mostly the same concepts with a different approach.

In *The Python Language Reference*, "3.2. The standard type hierarchy" presents the seven callable types, along with all the other built-in types.

The Python-3-only features discussed in this chapter have their own PEPs: PEP 3102 — Keyword-Only Arguments and PEP 3107 — Function Annotations.

For more about the current (as of mid-2014) use of annotations, two Stack Overflow questions are worth reading: "What are good uses for Python3's 'Function Annotations'" has a practical answer and insightful comments by Raymond Hettinger, and the answer for "What good are Python function annotations?" quotes extensively from Guido van Rossum.

PEP 362 — Function Signature Object is worth reading if you intend to use the `inspect` module that implements that feature.

A great introduction to functional programming in Python is A. M. Kuchling's Python Functional Programming HOWTO. The main focus of that text, however, is on the use of iterators and generators, which are the subject of Chapter 14.

`fn.py` is a package to support functional programming in Python 2 and 3. According to its author, Alexey Kachayev, `fn.py` provides "implementation of missing features to enjoy FP" in Python. It includes a `@recur.tco` decorator that implements tail-call optimization for unlimited recursion in Python, among many other functions, data structures, and recipes.

The StackOverflow question "Python: Why is functools.partial necessary?" has a highly informative (and funny) reply by Alex Martelli, author of the classic *Python in a Nutshell*.

Jim Fulton's Bobo was probably the first web framework that deserved to be called object-oriented. If you were intrigued by it and want to learn more about its modern rewrite, start at its Introduction. A little of the early history of Bobo appears in a comment by Phillip J. Eby in a discussion at Joel Spolsky's blog.

Soapbox

About Bobo

I owe my Python career to Bobo. I used it in my first Python web project in 1998. I discovered Bobo while looking for an object-oriented way to code web applications, after trying Perl and Java alternatives.

In 1997, Bobo had pioneered the object publishing concept: direct mapping from URLs to a hierarchy of objects, with no need to configure routes. I was hooked when I saw the beauty of this. Bobo also featured automatic HTTP query handling based on analysis of the signatures of the methods or functions used to handle requests.

Bobo was created by Jim Fulton, known as "The Zope Pope" thanks to his leading role in the development of the Zope framework, the foundation of the Plone CMS, SchoolTool, ERP5, and other large-scale Python projects. Jim is also the creator of ZODB—the Zope Object Database—a transactional object database that provides ACID (atomicity, consistency, isolation, and durability), designed for ease of use from Python.

Jim has since rewritten Bobo from scratch to support WSGI and modern Python (including Python 3). As of this writing, Bobo uses the `six` library to do the function introspection, in order to be compatible with Python 2 and Python 3 in spite of the changes in function objects and related APIs.

Is Python a Functional Language?

Around the year 2000, I was at a training in the United States when Guido van Rossum dropped by the classroom (he was not the instructor). In the Q&A that followed, somebody asked him which features of Python were borrowed from other languages. His answer: "Everything that is good in Python was stolen from other languages."

Shriram Krishnamurthi, professor of Computer Science at Brown University, starts his "Teaching Programming Languages in a Post-Linnaean Age" paper with this:

> Programming language "paradigms" are a moribund and tedious legacy of a bygone age. Modern language designers pay them no respect, so why do our courses slavishly adhere to them?

In that paper, Python is mentioned by name in this passage:

> What else to make of a language like Python, Ruby, or Perl? Their designers have no patience for the niceties of these Linnaean hierarchies; they borrow features as they wish, creating melanges that utterly defy characterization.

Krishnamurthi submits that instead of trying to classify languages in some taxonomy, it's more useful to consider them as aggregations of features.

Even if it was not Guido's goal, endowing Python with first-class functions opened the door to functional programming. In his post "Origins of Python's *Functional* Features", he says that `map`, `filter`, and `reduce` were the motivation for adding `lambda` to Python in the first place. All of these features were contributed together by Amrit Prem for Python 1.0 in 1994 (according to Misc/HISTORY in the CPython source code).

`lambda`, `map`, `filter`, and `reduce` first appeared in Lisp, the original functional language. However, Lisp does not limit what can be done inside a `lambda`, because everything in Lisp is an expression. Python uses a statement-oriented syntax in which expressions cannot contain statements, and many language constructs are statements—including `try/catch`, which is what I miss most often when writing `lambdas`. This is the price to pay for Python's highly readable syntax.[3] Lisp has many strengths, but readability is not one of them.

Ironically, stealing the list comprehension syntax from another functional language—Haskell—significantly diminished the need for `map` and `filter`, and also for `lambda`.

Besides the limited anonymous function syntax, the biggest obstacle to wider adoption of functional programming idioms in Python is the lack of tail-recursion elimination, an optimization that allows memory-efficient computation of a function that makes a recursive call at the "tail" of its body. In another blog post, "Tail Recursion Elimination", Guido gives several reasons why such optimization is not a good fit for Python. That post is a great read for the technical arguments, but even more so because the first three and most important reasons given are usability issues. It is no accident that Python is a pleasure to use, learn, and teach. Guido made it so.

So there you have it: Python is, by design, not a functional language—whatever that means. Python just borrows a few good ideas from functional languages.

The Problem with Anonymous Functions

Beyond the Python-specific syntax constraints, anonymous functions have a serious drawback in every language: they have no name.

I am only half joking here. Stack traces are easier to read when functions have names. Anonymous functions are a handy shortcut, people have fun coding with them, but sometimes they get carried away—especially if the language and environment encourage deep nesting of anonymous functions, like JavaScript on Node.js. Lots of nested anonymous functions make debugging and error handling hard. Asynchronous programming

3. There also the problem of lost indentation when pasting code to Web forums, but I digress.

in Python is more structured, perhaps because the limited `lambda` demands it. I promise to write more about asynchronous programming in the future, but this subject must be deferred to Chapter 18. By the way, promises, futures, and deferreds are concepts used in modern asynchronous APIs. Along with coroutines, they provide an escape from the so-called "callback hell." We'll see how callback-free asynchronous programming works in "From Callbacks to Futures and Coroutines" on page 562.