Chapter 16. Function Basics

In Part III, we studied basic procedural statements in Python. Here, we'll move on to explore a set of additional statements and expressions that we can use to create functions of our own.

In simple terms, a *function* is a package of code invoked by name. It labels and groups a set of statements so they can be run more than once in a program. A function also can compute a result value, and lets us specify parameters that serve as inputs and may differ each time the function's code is run. Wrapping an operation in a function makes it a generally useful tool, which we can apply in a variety of contexts.

On a more pragmatic level, functions are the alternative to programming by *cutting and pasting*—rather than having multiple redundant copies of an operation's code, we can factor it into a single function. In so doing, we reduce our future work radically: if the operation must be changed later, we have only one copy to update in the function, not many scattered throughout the program.

Functions are also the most basic program structure Python provides for maximizing code *reuse*, and lead us to the larger notions of program *design*. As you'll see, functions let us split complex systems into manageable parts. By implementing each part as a function, we make it both reusable and easier to code.

Table 16-1 abstractly previews the function-related tools we'll study in this part of the book—a set that includes call expressions, two ways to make functions (`def` and `lambda`), two ways to manage scope visibility (`global` and `nonlocal`), two ways to send results back to callers (`return` and `yield`), and tools to pause for results (`async` and `await`). While this comprises a feature-rich topic, you'll find that its commonly used core is straightforward.

Table 16-1. Function-related statements and expressions

Statement or expression	Examples
Call expressions	`myfunc('hack', tool=python, *versions)`
`def`	`def printer(message): print('Hello', message)`
`return`	`def adder(a, b=1, *c): return a + b + c[0]`
`lambda`	`funcs = [lambda x: x**2, lambda x: x**3]`
`global`	`x = 'old' def changer(): global x; x = 'new'`
`nonlocal`	`def outer(): x = 'old' def changer(): nonlocal x; x = 'new'`
`yield`	`def squares(x): for i in range(x): yield i ** 2`
Generator expressions	`(i ** 2 for i in range(x))`
`async/await`	`async def consumer(a, b): await producer(b)`
Decorators and annotations	`@tracer def func(a: 'hack' = None) -> None`

Why Use Functions?

Before we get into the details, let's establish a clearer picture of what functions are all about. Functions are a nearly universal program-structuring device. You may have come across them before in other languages, where they may have been called *subroutines* or *procedures*. As a brief introduction, functions serve two primary development roles, and serve as the basis of other tools:

Maximizing reuse and minimizing redundancy

As in most programming languages, Python functions are the simplest way to package logic you may wish to use in more than one place and more than one time. Up until now, almost all the code we've been

writing has run immediately. Functions allow us to defer and generalize code to be used arbitrarily many times later. Because they allow us to code an operation in a single place and use it in many others, functions are also a *factoring* tool: they enable us to reduce code redundancy in our programs, and thereby reduce maintenance effort.

Dividing and conquering

Functions also provide a tool for splitting systems into pieces that have well-defined roles and *manageable* scopes. For instance, to make a pizza from scratch, you would start by mixing the dough, rolling it out, adding toppings, baking it, and so on. If you were programming a pizza-making robot, functions would help you divide the overall "make pizza" task into smaller parts—one function for each subtask in the process. Because it's easier to implement the smaller tasks in isolation than it is to implement the entire process at once, this makes larger tasks more practical.

Implementing object methods

In general, functions are about *procedure*—how to do something, rather than what you're doing it to. Nevertheless, when paired with an implied subject, they can also be used to code object-specific behavior known as methods. You'll see why this distinction matters in Part VI, when we start making new objects with classes. As you'll find, classes are largely just packages of the functions you'll learn to code here.

In this part of the book, we'll explore the tools used to code functions in Python: function basics, scope rules, and argument passing, along with a few related concepts such as generators, coroutines, and functional tools. Because its importance begins to become more apparent at this level of coding, we'll also revisit the notion of polymorphism, which was introduced earlier in the book. As you'll see, functions don't imply much new syntax, but they do lead us to some bigger programming ideas.

Function Coding Overview

Although it wasn't made very formal, we've already *used* functions in earlier chapters. For instance, we called the built-in `open` function to make a file object, invoked the `len` built-in function to ask for the number of items in a collection object, and employed tools like `zip` and `range` for iteration tasks.

In this chapter, we will begin exploring how to write *new* functions in Python. Functions we write behave the same way as the built-ins we've already seen: they are called in expressions, are passed values, and return results. But writing new functions requires the application of a few additional tools and ideas that haven't yet been introduced. Moreover, functions behave very differently in Python than they do in compiled languages like C.

Basic Function Tools

As a road map and brief rundown on what we'll study in this part of the book, here are the basic components and concepts in Python's function toolbox:

- **`def` creates a function and assigns it to a name.** Python functions are coded with `def` statements. When Python reaches and runs a `def`, it generates a new function object and assigns it to the function's name. As with all assignments, the function name becomes a reference to the function object. There's nothing magical about the name of a function—as you'll see, the function object can be assigned to other names, stored in a list, and so on. Function objects may also have arbitrary user-defined *attributes* attached to them to record data.
- **`def` is executable code.** Unlike functions in compiled languages, `def` is an executable statement—your function does not exist until Python runs its `def`. In fact, it's legal (and even occasionally useful) to nest `def` statements inside `if` statements, `for` loops, and even other `def`s. In typical usage, `def` statements are coded at the top level of module files, and are automatically run to make functions when their modules are imported.
- **`return` sends a result object back to the caller.** When a function is called, the caller normally stops until the function finishes its work and returns control to the caller. Functions that compute a value send it back to the caller with a `return` statement, and the returned value becomes the result of the function call. A `return` without a value simply returns to the caller (and sends back `None`, the default result).
- **`lambda` creates a function but returns it as a result.** Function objects may also be created with the `lambda` expression, a feature that allows us to code *inline* function definitions in places where a `def` statement won't work syntactically. Like `def`, functions made this way are created when the `lambda` is reached and run. Because `lambda` is not typically coded at the top level of a module, though, such functions are made during

program runs, not imports. `lambda` is an optional convenience best seen as a sidebar to the `def` statement.

Advanced Function Tools

In addition, Python functions can leverage more advanced tools like the following, which we'll take up after we've introduced the basics:

- **`global` declares module-level variables that are to be assigned.** By default, all names assigned in a function are local to that function and exist only while the function runs. To assign a name in the enclosing module, functions need to list it in a `global` statement. More generally, names are always looked up in *scopes*—places where variables are stored—and names are bound to scopes per the location of assignments in your code.
- **`nonlocal` declares enclosing-function variables that are to be assigned.** In the same category as `global`, the `nonlocal` statement allows a function to assign a name that exists in the scope of a syntactically enclosing `def` statement. This allows enclosing functions to serve as a place to retain *state*—information remembered between function calls—without using shared global names.
- **`yield` sends a result object back to the caller, but remembers where it left off.** Functions known as *generators* may also use the `yield` statement to send back a value and suspend their state such that they may be resumed later, to produce a series of results over time. Along with their *generator expression* kin, such functions save space and avoid delays just like the built-in iterables we met in the prior part of this book.
- **`await/async` pause a waiting function so that other tasks may run.** Functions known as *coroutines* can suspend their execution until a required result is available. This is an advanced applications-level topic, but allows code to use language tools to get other work done while waiting on a blocking event like slow IO.

General Function Concepts

Finally, along the way we'll explore a handful of core concepts that span functions of all kinds:

- **Arguments are passed by assignment (object reference).** In Python, arguments are passed to functions by assignment—which, as we've

learned, means by object reference. As you'll see, in Python's model the caller and function share *objects* by references, but there is no name aliasing. Changing an argument name within a function does not also change the corresponding name in the caller, but changing passed-in mutable objects in place can change objects shared by the caller, and serve as a function result.

- **Arguments are passed by position, unless you say otherwise.** Values you pass in a function call match argument names in a function's definition from left to right by default. For flexibility, function *calls* can also pass arguments by name with `name=value` keyword syntax and unpack arbitrarily many arguments to send with `*args` and `**args` starred-value notation. Function *definitions* use the same syntax forms to specify argument defaults and collect arbitrarily many arguments received.
- **Arguments, return values, and variables are not declared.** As with everything in Python, there are no type constraints on functions. In fact, nothing about a function needs to be declared ahead of time: you can pass in arguments of any type, return any kind of object, and so on. As a consequence, a single function can often be applied to a variety of object types—any objects that have a compatible *interface* (support for expected methods and expressions) will do, regardless of their specific types. This makes code flexible by design.
- **Attributes, annotations, and decorators support advanced roles.** Functions can optionally be augmented with both user-defined attributes and other tools that extend their roles. Annotations of arguments and results, for example, can serve a variety of goals. Among these, *type hinting* (noted in Chapter 6) gives suggested object types but is unused by Python itself and simply serves as a weighty form of documentation, which we won't cover further in this book. More usefully, functions can also be prefixed with `@` decorators that add extra layers of logic but are used in ways that merit largely deferring until Part VI when we've also learned about classes.

If some of the preceding words didn't sink in, don't worry—we'll explore most of these concepts with real code in this part of the book. Let's get started by expanding on some of the basics by coding a few abstract examples.

def Statements

The `def` statement creates a function object and assigns it to a name. Its general format and usage is as follows:

```
def name(arg1, arg2,… argN):        # Define a functior
    statements                      # Function body

name(val1, val2,… valN)             # Call it later in
```

As with all compound statements, `def` consists of a *header* line followed by a block of other statements, usually indented (though a simple statement after the colon works as usual). The statement block becomes the function's *body*—that is, the code Python executes each time the function is later called.

The `def` header line specifies a function *name* that is assigned the new function object, along with a list of zero or more *arguments* (sometimes called *parameters*) enclosed in parentheses (even for zero arguments). The argument names in the header are *assigned* to the objects passed in parentheses at the point of call elsewhere in your code.

To *call* the function later in your program and run its body, you can use the function *name* assigned by `def` , passing in zero or more *values* (objects) to match the arguments listed in the `def` header. A call expression can also denote the function with other types of object references; it need not necessarily be the name coded in the `def` header.

return Statements

Function bodies often contain one or more `return` statements that make sense only inside a `def` :

```
def name(arg1, arg2,… argN):
    …
    return result                   # The result of the
```

The Python `return` statement can show up anywhere in a function body. When reached, it *ends* the function call and sends a *result* back to the caller to

serve as the result of the call expression. The `return` statement consists of an optional object-value expression that gives the result. If the value is omitted, `return` sends back a `None` by default.

The `return` statement itself is optional too; if it's not present, the function exits when the control flow falls off the end of the function body. Technically, a function *without* a `return` statement also returns the `None` object automatically, but this return value is usually ignored at the call, by using a call-expression *statement* of Chapter 11.

Functions may also contain `yield` statements used to produce a series of values over time, as well as `await` and `async for` / `with` statements used to suspend the function's execution, but we'll defer discussion of these more advanced tools until we survey generator and coroutine topics in Chapter 20.

def Executes at Runtime

The Python `def` is a true executable statement: when it runs, it creates a new function object and assigns it to a name. (Remember, all we have in standard Python is *runtime*; there is no such thing as a separate compile time.) Because it's a statement, a `def` can appear anywhere a statement can—even *nested* in other statements. For instance, although `def` s are normally top-level code run when the module enclosing them is imported, it's also completely legal to nest a function `def` inside an `if` statement to select between alternative definitions:

```
if test:
    def func():             # Define func this way
        …
else:
    def func():             # Or else this way
        …
…
func()                      # Call the version selected
```

One way to understand this code is to realize that the `def` is much like an `=` statement: it simply assigns a name (like `func` here) at runtime. Unlike in compiled languages such as C, Python functions do not need to be fully defined before the program runs. More generally, `def` s are not run until they

are reached, and the code *inside* `def` s is not run until the functions are later called.

Because function definition happens at runtime, there's nothing special about the function name. What's important is the object to which it refers. For example:

```
othername = func            # Assign function object
othername()                 # Call func again
```

Here, the function was assigned to a different name and called through the new name. Like everything else in Python, functions are just *objects*; they are recorded explicitly in memory at program execution time. In fact, besides calls, functions allow arbitrary *attributes* to be attached to record information for later use:

```
def func(): …               # Create + assign function c
func()                      # Call object via its name
func.attr = value           # Attach attributes
```

In other words, functions are *first-class objects*, to borrow a term introduced in Chapter 9. While they don't support some operations that other objects do, functions inhabit the same category as every object. As you'll see, passing them about and storing them in other objects is both syntactically legal and surprisingly useful.

lambda Makes Anonymous Functions

In addition to `def`, you can make a new function with the `lambda` expression. It's coded and might be used like this:

```
name = lambda arg1, arg2,… argN: expression

name(val1, val2,… valN)
```

Like `def`, `lambda` makes a new function object to be called later. It begins with the word `lambda`, followed by an arguments-list *header* that works the same as in `def` but is coded without parentheses. Unlike `def`, the code after

the `:` in `lambda` is a single *expression*—it cannot contain statements and is the *implied* body and return value of the function that `lambda` makes. We don't need to say `return` in a `lambda` (and we can't).

Also unlike `def`, `lambda` does not itself assign the new function to a name, but simply *returns* it as the result of the whole `lambda` expression. This snippet manually assigns the function to a name through which it is called, but that's optional: the result might also be saved in another object or passed for use elsewhere. Because of this, `lambda` is usually called an "anonymous" function—one that's unnamed.

You'll learn more about `lambda` later and see it in action along the way. But your first question may be, Why have a version of `def` whose code is limited to just one expression? In short, `lambda` can be used in places that `def` cannot, including in calls and object literals where we want to embed small bits of deferred and runnable code. While `def` handles larger tasks and also supports embedding functions by name, `lambda` is an optional amenity in simpler roles.

And if you're keeping track, we've now seen *four* expression equivalents for more general statements—the `lambda` for `def`, the `:=` named assignment for `=`, the `if`/`else` ternary for `if`, and the comprehension for `for`. Although lambdas are limited to expressions, they can use any of these expression equivalents to code assignments, logic, and loops. All these expression forms were accumulated over time, but are convenient alternatives in practical code.

A First Example: Definitions and Calls

Apart from their runtime flavor (which tends to seem most novel to programmers with backgrounds in compiled languages), Python functions are straightforward to use. Let's code a first real example to demonstrate the fundamentals, from both sides of the function equation: *definition* (the `def` or `lambda` that creates a function) and *calls* (the expressions that tell Python to run the function's body).

Definition

Here's a definition typed interactively that defines a function named `times`, which returns the product of its two arguments (it's not much, but it demos the basic bits). As usual "…" prompts are omitted here for copy and paste:

```
>>> def times(x, y):        # Create and assign functior
        return x * y         # Body executed when called
```

When Python reaches and runs this `def`, it creates a new function object that packages the function's code, and assigns this object to the name `times`. Typically, such a statement is coded in a module file and runs when the enclosing file is imported; for something this small, though, the interactive REPL suffices.

As an aside, a `lambda` can have the same effect if we assign its result to a name, though it's typically used in more focused roles than this, and often not as top-level code. Try the `def`, `lambda`, or both if you're working along—they both make function objects that work the same:

```
>>> times = lambda x, y: x * y
```

Calls

Both `def` and `lambda` make a function but do not call it. After either has run, you can *call* (i.e., run) the function in your program by adding parentheses after the function's name. The parentheses may optionally contain one or more object arguments, to be passed (i.e., assigned) to the names in the function's header:

```
>>> times(2, 4)             # Arguments in parentheses
8
```

This expression passes two arguments to `times`. As mentioned previously, arguments are passed by assignment, so in this case the name `x` in the function header is assigned the object `2`, `y` is assigned `4`, and the function's body is run.

For this function, the body is just a `def`'s `return` statement or a `lambda`'s expression, that sends back the result as the value of the call expression. The returned object was printed here interactively (as in most languages, `2 * 4` is `8` in Python), but if we needed to use it later we could instead assign it to a variable. For example:

```
>>> x = times(3.14, 4)      # Save the result object
>>> x
12.56
```

Now, watch what happens when the function is called a third time, with very different kinds of objects passed in:

```
>>> times('Py', 4)          # Functions are "typeless"
'PyPyPyPy'
```

This time, our function means something wholly different. In this third call, a string and an integer are passed to `x` and `y`, instead of two numbers. Recall that `*` works on both numbers and sequences; because we don't constrain the types of variables, arguments, or return values in Python, we can use `times` to either *multiply* numbers or *repeat* sequences.

In other words, what our `times` function means and does depends on what we pass into it. This is a core idea in Python (and perhaps the key to using the language well), which merits a separate callout here.

Polymorphism in Python

As we just saw, the very meaning of the expression `x * y` in our simple `times` function depends completely upon the kinds of objects that `x` and `y` are—thus, the same function can perform multiplication in one instance and repetition in another. Python leaves it up to the *objects* to do something reasonable for the syntax. Really, `*` is just a dispatch mechanism that routes control to the objects being processed.

This sort of type-dependent behavior is known as *polymorphism*, a term we first met in Chapter 4 that essentially means that the meaning of an operation depends on the objects being operated upon. Because it's a dynamically typed

language, polymorphism runs rampant in Python. In fact, *every* operation is a polymorphic operation in Python: printing, indexing, the `*` operator, and much more.

This is deliberate, and it accounts for much of the language's conciseness and flexibility. A single function, for instance, can generally be applied to a whole category of object types automatically. As long as those objects support the expected *interface* (a.k.a. protocol), the function can process them. That is, if the objects passed into a function have the expected methods and expression operators, they are plug-and-play compatible with the function's logic.

Even in our simple `times` function, this means that *any* two objects that support a `*` will work, no matter what they may be, and no matter when they are coded. This function will work on two numbers (performing multiplication), or a string and a number (performing repetition), or any other combination of objects supporting the expected interface—even class-based objects we have not even imagined yet.

Moreover, if the objects passed in do *not* support this expected interface, Python will detect the error when the `*` expression is run and raise an exception automatically. It's therefore usually pointless to code error checking in such code ourselves. In fact, doing so would limit our function's utility, as it would be restricted to work only on objects whose types we test for:

```
>>> times('not', 'quite')
TypeError: can't multiply sequence by non-int of type '
```

This turns out to be a crucial philosophical difference between Python and statically typed languages like C++ and Java: in Python, your code is *not supposed to care* about specific data types. If it does, it will be limited to working on just the types you anticipated when you wrote it, and it will not support other compatible object types coded now or in the future. Although it is possible to test for types with tools like the `type` built-in function, doing so breaks your code's flexibility. By and large, we code to object *interfaces* in Python, not data types.

Of course, some programs have unique requirements, and this polymorphic model of programming means we have to *test* our code to detect some errors that statically typed compiled languages might be able to detect earlier. In

exchange for an initial bit of extra testing, though, we radically reduce the amount of code we have to write, and radically increase our code's flexibility. As you'll come to find with time, it's a resounding net win in practice.

A Second Example: Intersecting Sequences

Next, let's explore a second function example that does something a bit more useful than multiplying arguments and further illustrates function basics.

In Chapter 13, we coded a `for` loop that collected items held in common in two strings (and called it nearly intersection, except for duplicates). We noted there that the code wasn't as useful as it could be because it was set up to work only on specific variables and could not be rerun later. Of course, we could copy the code and paste it into each place where it needs to be run, but this solution is neither good nor general—we'd still have to edit each copy to support different sequence names, and changing the algorithm would then require changing multiple copies.

Definition

By now, you can probably guess that the solution to this dilemma is to package the `for` loop inside a function. Doing so offers a number of advantages:

- Putting the code in a function makes it a tool that you can run as many times as you like.
- Because callers can pass in arbitrary arguments, functions are general enough to work on any two sequences (or other iterables) you wish to intersect.
- When the logic is packaged in a function, you have to change code in only *one* place if you ever need to change the way the intersection works.
- Coding the function in a module file means it can be imported and reused by any program run on your machine.

In effect, wrapping the code in a function makes it a general utility, as captured by Example 16-1.

Example 16-1. inter1.py

```
def intersect(seq1, seq2):
    res = []                        # Start empty
    for x in seq1:                  # Scan seq1
        if x in seq2:               # Common item?
            res.append(x)           # Add to end
    return res
```

The transformation from the simple code of Chapter 13 to this function is straightforward; we've just nested the original logic under a `def` header and made the objects on which it operates passed-in parameter names. Because this function computes a result, we've also added a `return` statement to send a result object back to the caller.

Calls

Before you can call a function, you have to make it. To do this, run its `def` statement, either by typing it interactively or by coding it in a module file and importing the file. You can paste it at a REPL as a whole, but we'll import it from a file here (per Chapter 3, make sure you can see it in your REPL by working in the file's folder if needed). Once you've run the `def` by such means, you can call the function by passing any two sequence objects in parentheses:

```
>>> from inter1 import intersect      # Get function fro
>>> s1 = 'HACK'
>>> s2 = 'CHOK'
>>> intersect(s1, s2)                 # Pass two strings
['H', 'C', 'K']
```

Here, we've passed in two strings, and we get back a list containing the characters in common. The algorithm the function uses is simple: "For every item in the first argument, if that item is also in the second argument, append the item to the result." It's a little shorter to say that in Python than in English, but it works out the same.

To be fair, our `intersect` function could probably be quicker (it executes naive nested loops), isn't really mathematical intersection (there may be

duplicates in the result), and isn't required at all (as we've seen, Python's set objects provide a built-in intersection operation kicked off with `&`). Indeed, the function could be replaced with a single list comprehension expression, as it exhibits the classic collector-loop code pattern:

```
>>> [x for x in s1 if x in s2]
['H', 'C', 'K']
```

Which also works as a loop inside a `lambda` body expression—despite being five lines in the `def`. This suffices as a demo, but doesn't make much sense in a simple role like this (again, we'll put lambdas to better use later):

```
>>> intersect = lambda seq1, seq2: [x for x in seq1 if
>>> intersect(s1, s2)
['H', 'C', 'K']
```

However it's coded, though, this function does the job: its single piece of code can apply to an entire range of object types, as the next section explains. In fact, we'll improve and extend this to support arbitrarily many operands in Chapter 18, after we uncover more about argument passing modes.

Polymorphism Revisited

Like all good functions in Python, `intersect` is polymorphic. That is, it works on arbitrary types, as long as they support the expected object interface:

```
>>> x = intersect([1, 2, 3], (1, 4))       # Pass mixed
>>> x                                      # Saved resul
[1]
```

This time, we passed in different types of objects to our function—a list and a tuple (mixed types)—and it still picked out the common items. Because you don't have to specify the types of arguments ahead of time, the `intersect` function happily iterates through any kind of objects you send it, as long as they support the expected interfaces.

For `intersect`, this means that the first argument has to support the `for` loop, and the second has to support the `in` membership test, in both `def`

and `lambda` flavors. Any two such objects will work, regardless of their specific types—that includes physically stored sequences like strings and lists; all the iterable objects we met in Chapter 14, including files and dictionaries; and even any class-based objects we code that apply operator overloading techniques we'll discuss later in the book.[1]

And here again, if we pass in objects that do *not* support these interfaces (e.g., numbers), Python will automatically detect the mismatch and raise an exception for us—which is exactly what we want, and the best we could do on our own if we coded explicit type tests:

```
>>> intersect([1, 2, 3], 1)
TypeError: argument of type 'int' is not iterable
```

By not coding type tests and allowing Python to detect the mismatches for us, we both reduce the amount of code we need to write, and avoid artificially limiting our code's scope.

Segue: Local Variables

Perhaps the most interesting part of this example, though, is its names. It turns out that the variable `res` inside the `def` version of `intersect` is what in Python is called a *local variable*—a name that is visible only to code inside the function `def` and that exists only while the function call runs. In fact, because all names *assigned* in any way inside a function are classified as local variables by default, nearly all the names in the `def` qualify:

- `res` is obviously assigned, so it is a local variable.
- Arguments are passed by assignment, so `seq1` and `seq2` are, too.
- The `for` loop assigns items to a variable, so the name `x` is also local.

All these local variables appear when the function is called and disappear when the function exits—the `return` statement at the end of `intersect` sends back the result *object*, but the *name* `res` goes away. The same goes for argument names in the `lambda`, though its compression hides its loop variable (sans a nested `:=` to export it).

Because of this, a function's variables don't clash with names elsewhere, but also won't remember values between calls. Although the object returned by a

function lives on, retaining other *state information* requires other techniques. To fully explore the notion of locals and state, though, we need to move on to the scopes coverage of the next chapter.

Chapter Summary

This chapter introduced the core ideas behind function definition—the syntax and operation of the `def` and `return` statements and `lambda` expression, the behavior of function call expressions, and the notion and benefits of polymorphism in Python functions. As we saw, a `def` and `lambda` are executable code that create a function object at runtime; when the function is later called, objects are passed into it by assignment (which means object reference in Python), and computed values are sent back by `return` in `def` and implicitly in `lambda`. We also began exploring the concepts of local variables and scopes, but saved the details for Chapter 17. First, though, a quick quiz.

Test Your Knowledge: Quiz

1. What is the point of coding functions?
2. At what time does Python create a function?
3. What does a `def` function return if it has no `return` statement in it?
4. When does the code nested inside the function definition run?
5. What's wrong with checking the types of objects passed into a function?

Test Your Knowledge: Answers

1. Functions are the most basic way of avoiding code *redundancy* in Python—factoring code into functions means that we have only one copy of an operation's code to update in the future. Functions are also the basic unit of code *reuse* in Python—wrapping code in functions makes it a reusable tool, callable in a variety of programs. Finally, functions allow us to *divide* a complex system into manageable parts, each of which may be developed individually. Functions are also used for object *methods*, but we're postponing this role until we study classes.
2. A function is created when Python reaches and runs a `def` statement, which creates a function object and assigns it to the function's name. This

normally happens when the enclosing module file is imported by another module (recall that imports run the code in a file from top to bottom, including any `def` s), but it can also occur when a `def` is typed interactively or nested in other statements, such as `if` s. Functions are also created when a `lambda` is reached and run, though this doesn't normally happen during an import operation.

3. A `def` function returns the `None` object by default if the control flow falls off the end of the function body without running into a `return` statement. Such functions are usually called with expression statements, as assigning their `None` results to variables is generally pointless. A `return` statement with no expression in it also returns `None` . A `lambda` implicitly returns its expression's result, so it has no default.
4. The function *body* (the code nested inside the function `def` statement or following the colon in `lambda`) is run when the function is later invoked with a call expression. The body runs anew each time the function is called.
5. Checking the types of objects passed into a function effectively breaks the function's *flexibility*, constraining the function to work on specific types only. Without such checks, the function would likely be able to process an entire array of object types—any objects that support the interface expected by the function will work. (The term *interface* means the set of methods and expression operators the function's code runs.)

[1] This code will always work if we intersect files' contents obtained with `file.readlines()` . It may not work to intersect lines in open input files directly, though, depending on the file object's implementation of the `in` operator or general iteration. Files must generally be rewound (e.g., with a `file.seek(0)` or another `open`) after they have been read to end-of-file once, and so are single-pass iterators. As you'll see in Chapter 30 when we study operator overloading, objects implement the `in` operator either by providing the specific `__contains__` method or by supporting the general iteration protocol with the `__iter__` or older `__getitem__` methods; classes can code these methods arbitrarily to define what iteration means for their data.