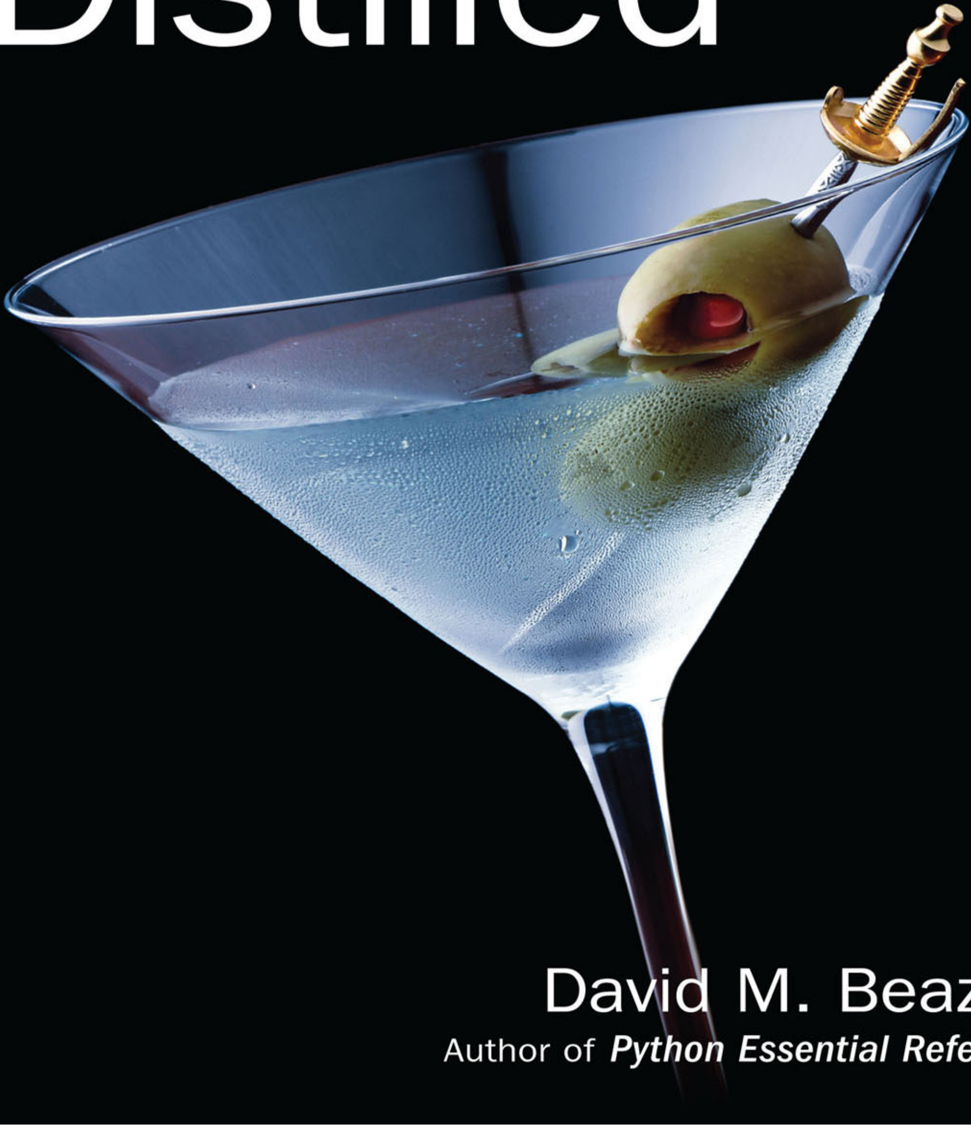
Python Distilled



David M. Beazley

Author of ***Python Essential Reference***

Python Distilled

This page intentionally left blank

Python Distilled

David M. Beazley

Addison-Wesley

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Library of Congress Control Number: 2021943288

ISBN-13: 978-0-13-417327-6
ISBN-10: 0-13-417327-9

ScoutAutomatedPrintCode

Contents

Preface

More than 20 years have passed since I authored the *Python Essential Reference*. At that time, Python was a much smaller language and it came with a useful set of batteries in its standard library. It was something that could mostly fit in your brain. The *Essential Reference* reflected that era. It was a small book that you could take with you to write some Python code on a desert island or inside a secret vault. Over the three subsequent revisions, the *Essential Reference* stuck with this vision of being a compact but complete language reference—because if you're going to code in Python on vacation, why not use all of it?

Today, more than a decade since the publication of the last edition, the Python world is much different. No longer a niche language, Python has become one of the most popular programming languages in the world. Python programmers also have a wealth of information at their fingertips in the form of advanced editors, IDEs, notebooks, web pages, and more. In fact, there's probably little need to consult a reference book when almost any reference material you might want can be conjured to appear before your eyes with the touch of a few keys.

If anything, the ease of information retrieval and the scale of the Python universe presents a different kind of challenge. If you're just starting to learn or need to solve a new problem, it can be overwhelming to know where to begin. It can also be difficult to separate the features of various tools from the core language itself. These kinds of problems are the rationale for this book.

Python Distilled is a book about programming in Python. It's not trying to document everything that's possible or has been done in Python. Its focus is on presenting a modern yet curated (or distilled) core of the language. It has been informed by my years of teaching Python to scientists, engineers, and software professionals. However, it's also a product of writing software libraries, pushing the edges of what makes Python tick, and finding out what's most useful.

For the most part, the book focuses on Python programming itself. This includes abstraction techniques, program structure, data, functions, objects, modules, and so forth—topics that will well serve programmers working on Python projects of any size. Pure reference material that can be easily obtained via an IDE (such as lists of functions, names of commands, arguments, etc.) is generally omitted. I've also made a conscious choice to not describe the fast-changing world of Python tooling—editors, IDEs, deployment, and related matters.

Perhaps controversially, I don't generally focus on language features related to large-scale software project management. Python is sometimes used for big and serious things—comprised of millions upon millions of lines of code. Such applications require specialized tooling, design, and features. They also involve committees, and meetings, and decisions to be made about very important matters. All this is too much for this small book. But perhaps the honest answer is that I don't use Python to write such applications—and neither should you. At least not as a hobby.

In writing a book, there is always a cut-off for the ever-evolving language features. This book was written during the era of Python 3.9. As such, it does not include some of the major additions planned for later releases—for example, structural pattern matching. That's a topic for a different time and place.

Last, but not least, I think it's important that programming remains fun. I hope that my book will not only help you become a productive Python programmer but also capture some of the magic that has inspired people to use Python for exploring the stars, flying helicopters on Mars, and spraying squirrels with a water cannon in the backyard.

Acknowledgments

I'd like to thank the technical reviewers, Shawn Brown, Sophie Tabac, and Pete Fein, for their helpful comments. I'd also like to thank my long-time editor Debra Williams Cauley for her work on this and past projects. The many students who have taken my classes have had a major if indirect impact on the topics covered in this book. Last, but not least, I'd like to thank Paula, Thomas, and Lewis for their support and love.

About the Author

David Beazley is the author of the *Python Essential Reference, Fourth Edition* (Addison-Wesley, 2010) and *Python Cookbook, Third Edition* (O'Reilly, 2013). He currently teaches advanced computer science courses through his company Dabeaz LLC (`www.dabeaz.com`). He's been using, writing, speaking, and teaching about Python since 1996.

5

Functions

Functions are a fundamental building block of most Python programs. This chapter describes function definitions, function application, scoping rules, closures, decorators, and other functional programming features. Particular attention is given to different programming idioms, evaluation models, and patterns associated with functions.

5.1 Function Definitions

Functions are defined with the `def` statement:

```
def add(x, y):
    return x + y
```

The first part of a function definition specifies the function name and parameter names that represent input values. The body of a function is a sequence of statements that execute when the function is called or applied. You apply a function to arguments by writing the function name followed by the arguments enclosed in parentheses: `a = add(3, 4)`. Arguments are fully evaluated left-to-right before executing the function body. For example, `add(1+1, 2+2)` is first reduced to `add(2, 4)` before calling the function. This is known as *applicative evaluation order*. The order and number of arguments must match the parameters given in the function definition. If a mismatch exists, a `TypeError` exception is raised. The structure of calling a function (such as the number of required arguments) is known as the function's call signature.

5.2 Default Arguments

You can attach default values to function parameters by assigning values in the function definition. For example:

```
def split(line, delimiter=','):
    statements
```

When a function defines a parameter with a default value, that parameter and all the parameters that follow it are optional. It is not possible to specify a parameter with no default value after any parameter with a default value.

Default parameter values are evaluated once when the function is first defined, not each time the function is called. This often leads to surprising behavior if mutable objects are used as a default:

```
def func(x, items=[]):
    items.append(x)
    return items

func(1)       # returns [1]
func(2)       # returns [1, 2]
func(3)       # returns [1, 2, 3]
```

Notice how the default argument retains the modifications made from previous invocations. To prevent this, it is better to use `None` and add a check as follows:

```
def func(x, items=None):
    if items is None:
        items = []
    items.append(x)
    return items
```

As a general practice, to avoid such surprises, only use immutable objects for default argument values—numbers, strings, Booleans, `None`, and so on.

5.3 Variadic Arguments

A function can accept a variable number of arguments if an asterisk (`*`) is used as a prefix on the last parameter name. For example:

```
def product(first, *args):
    result = first
    for x in args:
        result = result * x
    return result

product(10, 20)       # -> 200
product(2, 3, 4, 5)   # -> 120
```

In this case, all of the extra arguments are placed into the `args` variable as a tuple. You can then work with the arguments using the standard sequence operations—iteration, slicing, unpacking, and so on.

5.4 Keyword Arguments

Function arguments can be supplied by explicitly naming each parameter and specifying a value. These are known as keyword arguments. Here is an example:

```
def func(w, x, y, z):
    statements

# Keyword argument invocation
func(x=3, y=22, w='hello', z=[1, 2])
```

With keyword arguments, the order of the arguments doesn't matter as long as each required parameter gets a single value. If you omit any of the required arguments or if the name of a keyword doesn't match any of the parameter names in the function definition, a `TypeError` exception is raised. Keyword arguments are evaluated in the same order as they are specified in the function call.

Positional arguments and keyword arguments can appear in the same function call, provided that all the positional arguments appear first, values are provided for all nonoptional arguments, and no argument receives more than one value. Here's an example:

```
func('hello', 3, z=[1, 2], y=22)
func(3, 22, w='hello', z=[1, 2])   # TypeError. Multiple values for w
```

If desired, it is possible to force the use of keyword arguments. This is done by listing parameters after a `*` argument or just by including a single `*` in the definition. For example:

```
def read_data(filename, *, debug=False):
    ...

def product(first, *values, scale=1):
    result = first * scale
    for val in values:
        result = result * val
    return result
```

In this example, the `debug` argument to `read_data()` can only be specified by keyword. This restriction often improves code readability:

```
data = read_data('Data.csv', True)        # NO. TypeError
data = read_data('Data.csv', debug=True)  # Yes.
```

The `product()` function takes any number of positional arguments and an optional keyword-only argument. For example:

```
result = product(2,3,4)             # Result = 24
result = product(2,3,4, scale=10)   # Result = 240
```

5.5 Variadic Keyword Arguments

If the last argument of a function definition is prefixed with `**`, all the additional keyword arguments (those that don't match any of the other parameter names) are placed in a dictionary and passed to the function. The order of items in this dictionary is guaranteed to match the order in which keyword arguments were provided.

Arbitrary keyword arguments might be useful for defining functions that accept a large number of potentially open-ended configuration options that would be too unwieldy to list as parameters. Here's an example:

```
def make_table(data, **parms):
    # Get configuration parameters from parms (a dict)
    fgcolor = parms.pop('fgcolor', 'black')
    bgcolor = parms.pop('bgcolor', 'white')
    width = parms.pop('width', None)
    ...
    # No more options
    if parms:
        raise TypeError(f'Unsupported configuration options {list(parms)}')

make_table(items, fgcolor='black', bgcolor='white', border=1,
                  borderstyle='grooved', cellpadding=10,
                  width=400)
```

The `pop()` method of a dictionary removes an item from a dictionary, returning a possible default value if it's not defined. The `parms.pop('fgcolor', 'black')` expression used in this code mimics the behavior of a keyword argument specified with a default value.

5.6 Functions Accepting All Inputs

By using both `*` and `**`, you can write a function that accepts any combination of arguments. The positional arguments are passed as a tuple and the keyword arguments are passed as a dictionary. For example:

```
# Accept variable number of positional or keyword arguments
def func(*args, **kwargs):
    # args is a tuple of positional args
    # kwargs is dictionary of keyword args
    ...
```

This combined use of `*args` and `**kwargs` is commonly used to write wrappers, decorators, proxies, and similar functions. For example, suppose you have a function to parse lines of text taken from an iterable:

```
def parse_lines(lines, separator=',', types=(), debug=False):
    for line in lines:
        ...
        statements
        ...
```

Now, suppose you want to make a special-case function that parses data from a file specified by filename instead. To do that, you could write:

```
def parse_file(filename, *args, **kwargs):
    with open(filename, 'rt') as file:
        return parse_lines(file, *args, **kwargs)
```

The benefit of this approach is that the `parse_file()` function doesn't need to know anything about the arguments of `parse_lines()`. It accepts any extra arguments the caller provides and passes them along. This also simplifies the maintenance of the `parse_file()` function. For example, if new arguments are added to `parse_lines()`, those arguments will magically work with the `parse_file()` function too.

5.7 Positional-Only Arguments

Many of Python's built-in functions only accept arguments by position. You'll see this indicated by the presence of a slash (/) in the calling signature of a function shown by various help utilities and IDEs. For example, you might see something like `func(x, y, /)`. This means that all arguments appearing before the slash can only be specified by position. Thus, you could call the function as `func(2, 3)` but not as `func(x=2, y=3)`. For completeness, this syntax may also be used when defining functions. For example, you can write the following:

```
def func(x, y, /):
    pass

func(1, 2)      # Ok
func(1, y=2)    # Error
```

This definition form is rarely found in code since it was first supported only in Python 3.8. However, it can be a useful way to avoid potential name clashes between argument names. For example, consider the following code:

```
import time

def after(seconds, func, /, *args, **kwargs):
    time.sleep(seconds)
    return func(*args, **kwargs)
```

```
def duration(*, seconds, minutes, hours):
    return seconds + 60 * minutes + 3600 * hours

after(5, duration, seconds=20, minutes=3, hours=2)
```

In this code, `seconds` is being passed as a keyword argument, but it's intended to be used with the `duration` function that's passed to `after()`. The use of positional-only arguments in `after()` prevents a name clash with the `seconds` argument that appears first.

5.8 Names, Documentation Strings, and Type Hints

The standard naming convention for functions is to use lowercase letters with an underscore (_) used as a word separator—for example, `read_data()` and not `readData()`. If a function is not meant to be used directly because it's a helper or some kind of internal implementation detail, its name usually has a single underscore prepended to it—for example, `_helper()`. These are only conventions, however. You are free to name a function whatever you want as long as the name is a valid identifier.

The name of a function can be obtained via the `__name__` attribute. This is sometimes useful for debugging.

```
>>> def square(x):
...     return x * x
...
>>> square.__name__
'square'
>>>
```

It is common for the first statement of a function to be a documentation string describing its usage. For example:

```
def factorial(n):
    '''
    Computes n factorial. For example:

    >>> factorial(6)
    120
    >>>
    '''
    if n <= 1:
        return 1
    else:
        return n*factorial(n-1)
```

The documentation string is stored in the `__doc__` attribute of the function. It's often accessed by IDEs to provide interactive help.

Functions can also be annotated with type hints. For example:

```
def factorial(n: int) -> int:
    if n <= 1:
        return 1
    else:
        return n * factorial(n - 1)
```

The type hints don't change anything about how the function evaluates. That is, the presence of hints provides no performance benefits or extra runtime error checking. The hints are merely stored in the `__annotations__` attribute of the function which is a dictionary mapping argument names to the supplied hints. Third-party tools such as IDEs and code checkers might use the hints for various purposes.

Sometimes you will see type hints attached to local variables within a function. For example:

```
def factorial(n:int) -> int:
    result: int = 1           # Type-hinted local variable
    while n > 1:
        result *= n
        n -= 1
    return result
```

Such hints are completely ignored by the interpreter. They're not checked, stored, or even evaluated. Again, the purpose of the hints is to help third-party code-checking tools. Adding type hints to functions is not advised unless you are actively using code-checking tools that make use of them. It is easy to specify type hints incorrectly—and, unless you're using a tool that checks them, errors will go undiscovered until someone else decides to run a type-checking tool on your code.

5.9 Function Application and Parameter Passing

When a function is called, the function parameters are local names that get bound to the passed input objects. Python passes the supplied objects to the function "as is" without any extra copying. Care is required if mutable objects, such as lists or dictionaries, are passed. If changes are made, those changes are reflected in the original object. Here's an example:

```
def square(items):
    for i, x in enumerate(items):
        items[i] = x * x          # Modify items in-place
```

```
a = [1, 2, 3, 4, 5]
square(a)             # Changes a to [1, 4, 9, 16, 25]
```

Functions that mutate their input values, or change the state of other parts of the program behind the scenes, are said to have "side effects." As a general rule, side effects are best avoided. They can become a source of subtle programming errors as programs grow in size and complexity—it may not be obvious from reading a function call if a function has side effects or not. Such functions also interact poorly with programs involving threads and concurrency since side effects typically need to be protected by locks.

It's important to make a distinction between modifying an object and reassigning a variable name. Consider this function:

```
def sum_squares(items):
    items = [x*x for x in items]    # Reassign "items" name
    return sum(items)

a = [1, 2, 3, 4, 5]
result = sum_squares(a)
print(a)                            # [1, 2, 3, 4, 5]   (Unchanged)
```

In this example, it appears as if the `sum_squares()` function might be overwriting the passed `items` variable. Yes, the local `items` label is reassigned to a new value. But the original input value (`a`) is not changed by that operation. Instead, the local variable name `items` is bound to a completely different object—the result of the internal list comprehension. There is a difference between assigning a variable name and modifying an object. When you assign a value to a name, you're not overwriting the object that was already there—you're just reassigning the name to a different object.

Stylistically, it is common for functions with side effects to return `None` as a result. As an example, consider the `sort()` method of a list:

```
>>> items = [10, 3, 2, 9, 5]
>>> items.sort()                    # Observe: no return value
>>> items
[2, 3, 5, 9, 10]
>>>
```

The `sort()` method performs an in-place sort of list items. It returns no result. The lack of a result is a strong indicator of a side effect—in this case, the elements of the list got rearranged.

Sometimes you already have data in a sequence or a mapping that you'd like to pass to a function. To do this, you can use `*` and `**` in function invocations. For example:

```
def func(x, y, z):
    ...

s = (1, 2, 3)
```

```
# Pass a sequence as arguments
result = func(*s)

# Pass a mapping as keyword arguments
d = { 'x':1, 'y':2, 'z':3 }
result = func(**d)
```

You may be taking data from multiple sources or even supplying some of the arguments explicitly, and it will all work as long as the function gets all of its required arguments, there is no duplication, and everything in its calling signature aligns properly. You can even use `*` and `**` more than once in the same function call. If you're missing an argument or specify duplicate values for an argument, you'll get an error. Python will never let you call a function with arguments that don't satisfy its signature.

5.10 Return Values

The `return` statement returns a value from a function. If no value is specified or you omit the return statement, `None` is returned. To return multiple values, place them in a tuple:

```
def parse_value(text):
    '''
    Split text of the form name=val into (name, val)
    '''
    parts = text.split('=', 1)
    return (parts[0].strip(), parts[1].strip())
```

Values returned in a tuple can be unpacked to individual variables:

```
name, value = parse_value('url=http://www.python.org')
```

Sometimes named tuples are used as an alternative:

```
from typing import NamedTuple

class ParseResult(NamedTuple):
    name: str
    value: str

def parse_value(text):
    '''
    Split text of the form name=val into (name, val)
    '''
    parts = text.split('=', 1)
    return ParseResult(parts[0].strip(), parts[1].strip())
```

A named tuple works the same way as a normal tuple (you can perform all the same operations and unpacking), but you can also reference the returned values using named attributes:

```
r = parse_value('url=http://www.python.org')
print(r.name, r.value)
```

5.11 Error Handling

One problem with the `parse_value()` function in the previous section is error handling. What course of action should be taken if the input text is malformed and no correct result can be returned?

One approach is to treat the result as optional—that is, the function either works by returning an answer or returns `None` which is commonly used to indicate a missing value. For example, the function could be modified like this:

```
def parse_value(text):
    parts = text.split('=', 1)
    if len(parts) == 2:
        return ParseResult(parts[0].strip(), parts[1].strip())
    else:
        return None
```

With this design, the burden of checking for the optional result is placed on the caller:

```
result = parse_value(text)
if result:
    name, value = result
```

Or, in Python 3.8+, more compactly as follows:

```
if result := parse_value(text):
    name, value = result
```

Instead of returning `None`, you could treat malformed text as an error by raising an exception. For example:

```
def parse_value(text):
    parts = text.split('=', 1)
    if len(parts) == 2:
        return ParseResult(parts[0].strip(), parts[1].strip())
    else:
        raise ValueError('Bad value')
```

In this case, the caller is given the option of handling bad values with `try-except`. For example:

```
try:
    name, value = parse_value(text)
    ...
except ValueError:
    ...
```

The choice of whether or not to use an exception is not always clear-cut. As a general rule, exceptions are the more common way to handle an abnormal result. However, exceptions are also expensive if they frequently occur. If you're writing code where performance matters, returning `None`, `False`, `-1`, or some other special value to indicate failure might be better.

5.12 Scoping Rules

Each time a function executes, a local namespace is created. This namespace is an environment that contains the names and values of the function parameters as well as all variables that are assigned inside the function body. The binding of names is known in advance when a function is defined and all names assigned within the function body are bound to the local environment. All other names that are used but not assigned in the function body (the free variables) are dynamically found in the global namespace which is always the enclosing module where a function was defined.

There are two types of name-related errors that can occur during function execution. Looking up an undefined name of a free variable in the global environment results in a `NameError` exception. Looking up a local variable that hasn't been assigned a value yet results in an `UnboundLocalError` exception. This latter error is often a result of control flow bugs. For example:

```
def func(x):
    if x > 0:
        y = 42
    return x + y    # y not assigned if conditional is false

func(10)     # Returns 52
func(-10)    # UnboundLocalError: y referenced before assignment
```

`UnboundLocalError` is also sometimes caused by a careless use of in-place assignment operators. A statement such as `n += 1` is handled as `n = n + 1`. If used before `n` is assigned an initial value, it will fail.

```
def func():
    n += 1      # Error: UnboundLocalError
```

It's important to emphasize that variable names never change their scope—they are either global variables or local variables, and this is determined at function definition time. Here is an example that illustrates this:

```
x = 42
def func():
    print(x)     # Fails. UnboundLocalError
    x = 13

func()
```

In this example, it might look as though the `print()` function would output the value of the global variable `x`. However, the assignment of `x` that appears later marks `x` as a local variable. The error is a result of accessing a local variable that hasn't yet been assigned a value.

If you remove the `print()` function, you get code that looks like it might be reassigning the value of a global variable. For example, consider this:

```
x = 42
def func():
    x = 13
func()
# x is still 42
```

When this code executes, `x` retains its value of `42`, despite the appearance that it might be modifying the global variable `x` from inside the function `func`. When variables are assigned inside a function, they're always bound as local variables; as a result, the variable `x` in the function body refers to an entirely new object containing the value `13`, not the outer variable. To alter this behavior, use the `global` statement. `global` declares names as belonging to the global namespace, and it's necessary when a global variable needs to be modified. Here's an example:

```
x = 42
y = 37
def func():
    global x       # 'x' is in global namespace
    x = 13
    y = 0
func()
# x is now 13. y is still 37.
```

It should be noted that use of the `global` statement is usually considered poor Python style. If you're writing code where a function needs to mutate state behind the scenes, consider using a class definition and modify state by mutating an instance or class variable instead. For example:

```
class Config:
    x = 42

def func():
    Config.x = 13
```

Python allows nested function definitions. Here's an example:

```
def countdown(start):
   n = start
   def display():            # Nested function definition
       print('T-minus', n)
   while n > 0:
       display()
       n -= 1
```

Variables in nested functions are bound using lexical scoping. That is, names are resolved first in the local scope and then in successive enclosing scopes from the innermost scope to the outermost scope. Again, this is not a dynamic process—the binding of names is determined once at function definition time based on syntax. As with global variables, inner functions can't reassign the value of a local variable defined in an outer function. For example, this code does not work:

```
def countdown(start):
   n = start
   def display():
       print('T-minus', n)
   def decrement():
       n -= 1              # Fails: UnboundLocalError
   while n > 0:
        display()
        decrement()
```

To fix this, you can declare `n` as `nonlocal` like this:

```
def countdown(start):
   n = start
   def display():
       print('T-minus', n)
   def decrement():
       nonlocal n
       n -= 1              # Modifies the outer n
   while n > 0:
        display()
        decrement()
```

`nonlocal` cannot be used to refer to a global variable—it must reference a local variable in an outer scope. Thus, if a function is assigning to a global, you should still use the `global` declaration as previously described.

Use of nested functions and `nonlocal` declarations is not a common programming style. For example, inner functions have no outside visibility, which can complicate testing and debugging. Nevertheless, nested functions are sometimes useful for breaking complex calculations into smaller parts and hiding internal implementation details.

5.13 Recursion

Python supports recursive functions. For example:

```
def sumn(n):
    if n == 0:
        return 0
    else:
        return n + sumn(n-1)
```

However, there is a limit on the depth of recursive function calls. The function `sys.getrecursionlimit()` returns the current maximum recursion depth, and the function `sys.setrecursionlimit()` can be used to change the value. The default value is 1000. Although it is possible to increase the value, programs are still limited by the stack size enforced by the host operating system. When the recursion depth limit is exceeded, a `RuntimeError` exception is raised. If the limit is increased too much, Python might crash with a segmentation fault or another operating system error.

In practice, issues with the recursion limit only arise when you work with deeply nested recursive data structures such as trees and graphs. Many algorithms involving trees naturally lend themselves to recursive solutions—and, if your data structure is too large, you might blow the stack limit. However, there are some clever workarounds; see Chapter 6 on generators for an example.

5.14 The `lambda` Expression

An anonymous—unnamed—function can be defined with a `lambda` expression:

```
lambda args: expression
```

`args` is a comma-separated list of arguments, and `expression` is an expression involving those arguments. Here's an example:

```
a = lambda x, y: x + y
r = a(2, 3)              # r gets 5
```

The code defined with `lambda` must be a valid expression. Multiple statements, or nonexpression statements such as `try` and `while`, cannot appear in a `lambda` expression. `lambda` expressions follow the same scoping rules as functions.

One of the main uses of `lambda` is to define small callback functions. For example, you may see it used with built-in operations such as `sorted()`. For example:

```
# Sort a list of words by the number of unique letters
result = sorted(words, key=lambda word: len(set(word)))
```

Caution is required when a `lambda` expression contains free variables (not specified as parameters). Consider this example:

```
x = 2
f = lambda y: x * y
x = 3
g = lambda y: x * y
print(f(10))                # --> prints 30
print(g(10))                # --> prints 30
```

In this example, you might expect the call `f(10)` to print `20`, reflecting the fact that `x` was `2` at the time of definition. However, this is not the case. As a free variable, the evaluation of `f(10)` uses whatever value `x` happens to have at the time of evaluation. It could be different from the value it had when the `lambda` function was defined. Sometimes this behavior is referred to as *late binding*.

If it's important to capture the value of a variable at the time of definition, use a default argument:

```
x = 2
f = lambda y, x=x: x * y
x = 3
g = lambda y, x=x: x * y
print(f(10))                # --> prints 20
print(g(10))                # --> prints 30
```

This works because default argument values are only evaluated at the time of function definition and thus would capture the current value of `x`.

5.15 Higher-Order Functions

Python supports the concept of *higher-order functions*. This means that functions can be passed as arguments to other functions, placed in data structures, and returned by a function as a result. Functions are said to be *first-class objects*, meaning there is no difference between how you might handle a function and any other kind of data. Here is an example of a function that accepts another function as input and calls it after a time delay—for example, to emulate the performance of a microservice in the cloud:

```
import time

def after(seconds, func):
    time.sleep(seconds)
    func()

# Example usage
def greeting():
    print('Hello World')

after(10, greeting)       # Prints 'Hello World' after 10 seconds
```

Here, the `func` argument to `after()` is an example of what's known as a *callback function*. This refers to the fact that the `after()` function "calls back" to the function supplied as an argument.

When a function is passed as data, it implicitly carries information related to the environment in which the function was defined. For example, suppose the `greeting()` function makes use of a variable like this:

```
def main():
    name = 'Guido'
    def greeting():
        print('Hello', name)
    after(10, greeting)         # Produces: 'Hello Guido'

main()
```

In this example, the variable `name` is used by `greeting()`, but it's a local variable of the outer `main()` function. When `greeting` is passed to `after()`, the function remembers its environment and uses the value of the required `name` variable. This relies on a feature known as a *closure*. A closure is a function along with an environment containing all of the variables needed to execute the function body.

Closures and nested functions are useful when you write code based on the concept of lazy or delayed evaluation. The `after()` function, shown above, is an illustration of this concept. It receives a function that is not evaluated right away—that only happens at some later point in time. This is a common programming pattern that arises in other contexts. For example, a program might have functions that only execute in response to events—key presses, mouse movement, arrival of network packets, and so on. In all of these cases, function evaluation is deferred until something interesting happens. When the function finally executes, a closure ensures that the function gets everything that it needs.

You can also write functions that create and return other functions. For example:

```
def make_greeting(name):
    def greeting():
        print('Hello', name)
    return greeting

f = make_greeting('Guido')
g = make_greeting('Ada')

f()     # Produces: 'Hello Guido'
g()     # Produces: 'Hello Ada'
```

In this example, the `make_greeting()` function doesn't carry out any interesting computations. Instead, it creates and returns a function `greeting()` that does the actual work. That only happens when that function gets evaluated later.

In this example, the two variables `f` and `g` hold two different versions of the `greeting()` function. Even though the `make_greeting()` function that created those functions is no longer executing, the `greeting()` functions still remember the `name` variable that was defined—it's part of each function's closure.

One caution about closures is that binding to variable names is not a "snapshot" but a dynamic process—meaning the closure points to the `name` variable and the value that it was most recently assigned. This is subtle, but here's an example that illustrates where trouble can arise:

```
def make_greetings(names):
    funcs = []
    for name in names:
        funcs.append(lambda: print('Hello', name))
    return funcs

# Try it
a, b, c = make_greetings(['Guido', 'Ada', 'Margaret'])
a()    # Prints 'Hello Margaret'
b()    # Prints 'Hello Margaret'
c()    # Prints 'Hello Margaret'
```

In this example, a list of different functions is made (using `lambda`). It may appear as if they are all using a unique value of `name`, as it changes on each iteration of a `for` loop. This is not the case. All functions end up using the same value of `name`—the value it has when the outer `make_greetings()` function returns.

This is probably unexpected and not what you want. If you want to capture a copy of a variable, capture it as a default argument, as previously described:

```
def make_greetings(names):
    funcs = []
    for name in names:
        funcs.append(lambda name=name: print('Hello', name))
    return funcs

# Try it
a, b, c = make_greetings(['Guido', 'Ada', 'Margaret'])
a()    # Prints 'Hello Guido'
b()    # Prints 'Hello Ada'
c()    # Prints 'Hello Margaret'
```

In the last two examples, functions have been defined using `lambda`. This is often used as a shortcut for creating small callback functions. However, it's not a strict requirement. You could have rewritten it like this:

```
def make_greetings(names):
    funcs = []
    for name in names:
        def greeting(name=name):
            print('Hello', name)
        funcs.append(greeting)
    return funcs
```

The choice of when and where to use `lambda` is one of personal preference and a matter of code clarity. If it makes code harder to read, perhaps it should be avoided.

5.16 Argument Passing in Callback Functions

One challenging problem with callback functions is that of passing arguments to the supplied function. Consider the `after()` function written earlier:

```
import time

def after(seconds, func):
    time.sleep(seconds)
    func()
```

In this code, `func()` is hardwired to be called with no arguments. If you want to pass extra arguments, you're out of luck. For example, you might try this:

```
def add(x, y):
    print(f'{x} + {y} -> {x+y}')
    return x + y

after(10, add(2, 3))     # Fails: add() called immediately
```

In this example, the `add(2, 3)` function runs immediately, returning `5`. The `after()` function then crashes 10 seconds later as it tries to execute `5()`. That is definitely not what you intended. Yet there seems to be no obvious way to make it work if `add()` is called with its desired arguments.

This problem hints towards a greater design issue concerning the use of functions and functional programming in general—function composition. When functions are mixed together in various ways, you need to think about how function inputs and outputs connect together. It is not always simple.

In this case, one solution is to package up computation into a zero-argument function using `lambda`. For example:

```
after(10, lambda: add(2, 3))
```

A small zero-argument function like this is sometimes known as a *thunk*. Basically, it's an expression that will be evaluated later when it's eventually called as a zero-argument function. This can be a general-purpose way to delay the evaluation of any expression to a later point in time: put the expression in a `lambda` and call the function when you actually need the value.

As an alternative to using `lambda`, you could use `functools.partial()` to create a partially evaluated function like this:

```
from functools import partial

after(10, partial(add, 2, 3))
```

`partial()` creates a callable where one or more of the arguments have already been specified and are cached. It can be a useful way to make nonconforming functions match expected calling signatures in callbacks and other applications. Here are a few more examples of using `partial()`:

```
def func(a, b, c, d):
    print(a, b, c, d)

f = partial(func, 1, 2)       # Fix a=1, b=2
f(3, 4)                       # func(1, 2, 3, 4)
f(10, 20)                     # func(1, 2, 10, 20)

g = partial(func, 1, 2, d=4)  # Fix a=1, b=2, d=4
g(3)                          # func(1, 2, 3, 4)
g(10)                         # func(1, 2, 10, 4)
```

`partial()` and `lambda` can be used for similar purposes, but there is an important semantic distinction between the two techniques. With `partial()`, the arguments are evaluated and bound at the time the partial function is first defined. With a zero-argument `lambda`, the arguments are evaluated and bound when the `lambda` function actually executes later (the evaluation of everything is delayed). To illustrate:

```
>>> def func(x, y):
...     return x + y
...
>>> a = 2
>>> b = 3
>>> f = lambda: func(a, b)
>>> g = partial(func, a, b)
>>> a = 10
>>> b = 20
>>> f()     # Uses current values of a, b
30
>>> g()     # Uses initial values of a, b
5
>>>
```

Since partials are fully evaluated, the callables created by `partial()` are objects that can be serialized into bytes, saved in files, and even transmitted across network connections (for example, using the `pickle` standard library module). This is not possible with a `lambda` function. Thus, in applications where functions are passed around, possibly to Python interpreters running in different processes or on different machines, you'll find `partial()` to be a bit more adaptable.

As an aside, partial function application is closely related to a concept known as *currying*. Currying is a functional programming technique where a multiple-argument function is expressed as a chain of nested single-argument functions. Here is an example:

```
# Three-argument function
def f(x, y, z):
    return x + y + z

# Curried version
def fc(x):
    return lambda y: (lambda z: x + y + z)

# Example use
a = f(2, 3, 4)     # Three-argument function
b = fc(2)(3)(4)    # Curried version
```

This is not a common Python programming style and there are few practical reasons for doing it. However, sometimes you'll hear the word "currying" thrown about in conversations with coders who've spent too much time warping their brains with things like `lambda` calculus. This technique of handling multiple arguments is named in honor of the famous logician Haskell Curry. Knowing what it is might be useful—should you stumble into a group of functional programmers having a heated flamewar at a social event.

Getting back to the original problem of argument passing, another option for passing arguments to a callback function is to accept them separately as arguments to the outer calling function. Consider this version of the `after()` function:

```
def after(seconds, func, *args):
    time.sleep(seconds)
    func(*args)

after(10, add, 2, 3)   # Calls add(2, 3) after 10 seconds
```

You will notice that passing keyword arguments to `func()` is not supported. This is by design. One issue with keyword arguments is that the argument names of the given function might clash with argument names already in use (that is, `seconds` and `func`). Keyword arguments might also be reserved for specifying options to the `after()` function itself. For example:

```
def after(seconds, func, *args, debug=False):
    time.sleep(seconds)
    if debug:
        print('About to call', func, args)
    func(*args)
```

All is not lost, however. If you need to specify keyword arguments to `func()`, you can still do it using `partial()`. For example:

```
after(10, partial(add, y=3), 2)
```

If you wanted the `after()` function to accept keyword arguments, a safe way to do it might be to use positional-only arguments. For example:

```
def after(seconds, func, debug=False, /, *args, **kwargs):
    time.sleep(seconds)
    if debug:
        print('About to call', func, args, kwargs)
    func(*args, **kwargs)

after(10, add, 2, y=3)
```

Another possibly unsettling insight is that `after()` actually represents two different function calls merged together. Perhaps the problem of passing arguments can be decomposed into two functions like this:

```
def after(seconds, func, debug=False):
    def call(*args, **kwargs):
        time.sleep(seconds)
        if debug:
            print('About to call', func, args, kwargs)
        func(*args, **kwargs)
    return call

after(10, add)(2, y=3)
```

Now, there are no conflicts whatsoever between the arguments to `after()` and the arguments to `func`. However, there is a chance that doing this will introduce a conflict between you and your coworkers.

5.17 Returning Results from Callbacks

Another problem not addressed in the previous section is that of returning the results of the calculation. Consider this modified `after()` function:

```
def after(seconds, func, *args):
    time.sleep(seconds)
    return func(*args)
```

This works, but there are some subtle corner cases that arise from the fact that two separate functions are involved—the `after()` function itself and the supplied callback `func`.

One issue concerns exception handling. For example, try these two examples:

```
after("1", add, 2, 3)  # Fails: TypeError (integer is expected)
after(1, add, "2", 3)  # Fails: TypeError (can't concatenate int to str)
```

A `TypeError` is raised in both cases, but it's for very different reasons and in different functions. The first error is due to a problem in the `after()` function itself: A bad argument is being given to `time.sleep()`. The second error is due to a problem with the execution of the callback function `func(*args)`.

If it's important to distinguish between these two cases, there are a few options for that. One option is to rely on chained exceptions. The idea is to package errors from the callback in a different way that allows them to be handled separately from other kinds of errors. For example:

```
class CallbackError(Exception):
    pass

def after(seconds, func, *args):
    time.sleep(seconds)
    try:
        return func(*args)
    except Exception as err:
        raise CallbackError('Callback function failed') from err
```

This modified code isolates errors from the supplied callback into its own exception category. Use it like this:

```
try:
    r = after(delay, add, x, y)
except CallbackError as err:
    print("It failed. Reason", err.__cause__)
```

If there was a problem with the execution of `after()` itself, that exception would propagate out, uncaught. On the other hand, problems related to the execution of the supplied callback function would be caught and reported as a `CallbackError`. All of this is quite subtle, but in practice, managing errors is hard. This approach makes the attribution of blame more precise and the behavior of `after()` easier to document. Specifically, if there is a problem in the callback, it's always reported as a `CallbackError`.

Another option is to package the result of the callback function into some kind of result instance that holds both a value and an error. For example, define a class like this:

```
class Result:
    def __init__(self, value=None, exc=None):
        self._value = value
        self._exc = exc
    def result(self):
        if self._exc:
            raise self._exc
        else:
            return self._value
```

Then, use this class to return results from the after() function:

```
def after(seconds, func, *args):
    time.sleep(seconds)
    try:
        return Result(value=func(*args))
    except Exception as err:
        return Result(exc=err)

# Example use:

r = after(1, add, 2, 3)
print(r.result())           # Prints 5

s = after("1", add, 2, 3)   # Immediately raises TypeError. Bad sleep() arg.

t = after(1, add, "2", 3)   # Returns a "Result"
print(t.result())           # Raises TypeError
```

This second approach works by deferring the result reporting of the callback function to a separate step. If there is a problem with after(), it gets reported immediately. If there is a problem with the callback func(), that gets reported when a user tries to obtain the result by calling the result() method.

This style of boxing a result into a special instance to be unwrapped later is an increasingly common pattern found in modern programming languages. One reason for its use is that it facilitates type checking. For example, if you were to put a type hint on after(), its behavior is fully defined—it always returns a Result and nothing else:

```
def after(seconds, func, *args) -> Result:
    ...
```

Although it's not so common to see this kind of pattern in Python code, it does arise with some regularity when working with concurrency primitives such as threads and processes. For example, instances of a so-called Future behave like this when working with thread pools. For example:

```
from concurrent.futures import ThreadPoolExecutor

pool = ThreadPoolExecutor(16)
r = pool.submit(add, 2, 3)     # Returns a Future
print(r.result())              # Unwrap the Future result
```

5.18 Decorators

A decorator is a function that creates a wrapper around another function. The primary purpose of this wrapping is to alter or enhance the behavior of the object being wrapped. Syntactically, decorators are denoted using the special `@` symbol as follows:

```
@decorate
def func(x):
    ...
```

The preceding code is shorthand for the following:

```
def func(x):
    ...
func = decorate(func)
```

In the example, a function `func()` is defined. However, immediately after its definition, the function object itself is passed to the function `decorate()`, which returns an object that replaces the original `func`.

As an example of a concrete implementation, here is a decorator `@trace` that adds debugging messages to a function:

```
def trace(func):
    def call(*args, **kwargs):
        print('Calling', func.__name__)
        return func(*args, **kwargs)
    return call

# Example use
@trace
def square(x):
    return x * x
```

In this code, `trace()` creates a wrapper function that writes some debugging output and then calls the original function object. Thus, if you call `square()`, you will see the output of the `print()` function in the wrapper.

If only it were so easy! In practice, functions also contain metadata such as the function name, doc string, and type hints. If you put a wrapper around a function, this information gets hidden. When writing a decorator, it's considered best practice to use the `@wraps()` decorator as shown in this example:

```
from functools import wraps

def trace(func):
    @wraps(func)
    def call(*args, **kwargs):
        print('Calling', func.__name__)
        return func(*args, **kwargs)
    return call
```

The `@wraps()` decorator copies various function metadata to the replacement function. In this case, metadata from the given function `func()` is copied to the returned wrapper function `call()`.

When decorators are applied, they must appear on their own line immediately prior to the function. More than one decorator can be applied. Here's an example:

```
@decorator1
@decorator2
def func(x):
     pass
```

In this case, the decorators are applied as follows:

```
def func(x):
    pass

func = decorator1(decorator2(func))
```

The order in which decorators appear might matter. For example, in class definitions, decorators such as `@classmethod` and `@staticmethod` often have to be placed at the outermost level. For example:

```
class SomeClass(object):
    @classmethod                # Yes
    @trace
    def a(cls):
        pass

    @trace                      # No. Fails.
    @classmethod
    def b(cls):
        pass
```

The reason for this placement restriction has to do with the values returned by `@classmethod`. Sometimes a decorator returns an object that's different than a normal function. If the outermost decorator isn't expecting this, things can break. In this case, `@classmethod` creates a `classmethod` descriptor object (see Chapter 7). Unless the `@trace` decorator was written to account for this, it will fail if decorators are listed in the wrong order.

A decorator can also accept arguments. Suppose you want to change the `@trace` decorator to allow for a custom message like this:

```
@trace("You called {func.__name__}")
def func():
    pass
```

If arguments are supplied, the semantics of the decoration process is as follows:

```
def func():
    pass

# Create the decoration function
temp = trace("You called {func.__name__}")

# Apply it to func
func = temp(func)
```

In this case, the outermost function that accepts the arguments is responsible for creating a decoration function. That function is then called with the function to be decorated to obtain the final result. Here's what the decorator implementation might look like:

```
from functools import wraps

def trace(message):
    def decorate(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            print(message.format(func=func))
            return func(*args, **kwargs)
        return wrapper
    return decorate
```

One interesting feature of this implementation is that the outer function is actually a kind of a "decorator factory." Suppose you found yourself writing code like this:

```
@trace('You called {func.__name__}')
def func1():
    pass

@trace('You called {func.__name__}')
def func2():
    pass
```

That would quickly get tedious. You could simplify it by calling the outer decorator function once and reusing the result like this:

```
logged = trace('You called {func.__name__}')

@logged
def func1():
    pass
```

```
@logged
def func2():
    pass
```

Decorators don't necessarily have to replace the original function. Sometimes a decorator merely performs an action such as registration. For example, if you are building a registry of event handlers, you could define a decorator that works like this:

```
@eventhandler('BUTTON')
def handle_button(msg):
     ...
@eventhandler('RESET')
def handle_reset(msg):
     ...
```

Here's a decorator that manages it:

```
# Event handler decorator
_event_handlers = { }
def eventhandler(event):
     def register_function(func):
          _event_handlers[event] = func
          return func
     return register_function
```

5.19 Map, Filter, and Reduce

Programmers familiar with functional languages often inquire about common list operations such as map, filter, and reduce. Much of this functionality is provided by list comprehensions and generator expressions. For example:

```
def square(x):
    return x * x

nums = [1, 2, 3, 4, 5]
squares = [ square(x) for x in nums ]    # [1, 4, 9, 16, 25]
```

Technically, you don't even need the short one-line function. You could write:

```
squares = [ x * x for x in nums ]
```

Filtering can also be performed with a list comprehension:

```
a = [ x for x in nums if x > 2 ]    # [3, 4, 5]
```

If you use a generator expression, you'll get a generator that produces the results incrementally through iteration. For example:

```
squares = (x*x for x in nums)    # Creates a generator
for n in squares:
    print(n)
```

Python provides a built-in `map()` function that is the same as mapping a function with a generator expression. For example, the above example could be written:

```
squares = map(lambda x: x*x, nums)
for n in squares:
    print(n)
```

The built-in `filter()` function creates a generator that filters values:

```
for n in filter(lambda x: x > 2, nums):
    print(n)
```

If you want to accumulate or reduce values, you can use `functools.reduce()`. For example:

```
from functools import reduce
total = reduce(lambda x, y: x + y, nums)
```

In its general form, `reduce()` accepts a two-argument function, an iterable, and an initial value. Here are a few examples:

```
nums = [1, 2, 3, 4, 5]
total = reduce(lambda x, y: x + y, nums)         # 15
product = reduce(lambda x, y: x * y, nums, 1)    # 120

pairs = reduce(lambda x, y: (x, y), nums, None)
# (((((None, 1), 2), 3), 4), 5)
```

`reduce()` accumulates values left-to-right on the supplied iterable. This is known as a left-fold operation. Here is pseudocode for `reduce(func, items, initial)`:

```
def reduce(func, items, initial):
    result = initial
    for item in items:
        result = func(result, item)
    return result
```

Using `reduce()` in practice may be confusing. Moreover, common reduction operations such as `sum()`, `min()`, and `max()` are already built-in. Your code will be easier to follow (and likely run faster) if you use one of those instead of trying to implement common operations with `reduce()`.

5.20 Function Introspection, Attributes, and Signatures

As you have seen, functions are objects—which means they can be assigned to variables, placed in data structures, and used in the same way as any other kind of data in a program. They can also be inspected in various ways. Table 5.1 shows some common attributes of functions. Many of these attributes are useful in debugging, logging, and other operations involving functions.

Table 5.1 Function Attributes

Attribute	Description
`f.__name__`	Function name
`f.__qualname__`	Fully qualified name (if nested)
`f.__module__`	Name of module in which defined
`f.__doc__`	Documentation string
`f.__annotations__`	Type hints
`f.__globals__`	Dictionary that is the global namespace
`f.__closure__`	Closure variables (if any)
`f.__code__`	Underlying code object

The `f.__name__` attribute contains the name that was used when defining a function. `f.__qualname__` is a longer name that includes additional information about the surrounding definition environment.

The `f.__module__` attribute is a string that holds the module name in which the function was defined. The `f.__globals__` attribute is a dictionary that serves as the global namespace for the function. It is normally the same dictionary that's attached to the associated module object.

`f.__doc__` holds the function documentation string. The `f.__annotations__` attribute is a dictionary that holds type hints, if any.

`f.__closure__` holds references to the values of closure variables for nested functions. These are a bit buried, but the following example shows how to view them:

```
def add(x, y):
    def do_add():
        return x + y
    return do_add

>>> a = add(2, 3)
>>> a.__closure__
(<cell at 0x10edf1e20: int object at 0x10ecc1950>,
 <cell at 0x10edf1d90: int object at 0x10ecc1970>)
>>> a.__closure__[0].cell_contents
```

```
2
>>>
```

The `f.__code__` object represents the compiled interpreter bytecode for the function body.

Functions can have arbitrary attributes attached to them. Here's an example:

```
def func():
    statements

func.secure = 1
func.private = 1
```

Attributes are not visible within the function body—they are not local variables and do not appear as names in the execution environment. The main use of function attributes is to store extra metadata. Sometimes frameworks or various metaprogramming techniques utilize function tagging—that is, attaching attributes to functions. One example is the `@abstractmethod` decorator that's used on methods within abstract base classes. All that decorator does is attach an attribute:

```
def abstractmethod(func):
    func.__isabstractmethod__ = True
    return func
```

Some other bit of code (in this case, a metaclass) looks for this attribute and uses it to add extra checks to instance creation.

If you want to know more about a function's parameters, you can obtain its signature using the `inspect.signature()` function:

```
import inspect

def func(x: int, y:float, debug=False) -> float:
    pass

sig = inspect.signature(func)
```

Signature objects provide many convenient features for printing and obtaining detailed information about the parameters. For example:

```
# Print out the signature in a nice form
print(sig)  # Produces (x: int, y: float, debug=False) -> float

# Get a list of argument names
print(list(sig.parameters))   # Produces [ 'x', 'y', 'debug']

# Iterate over the parameters and print various metadata
for p in sig.parameters.values():
    print('name', p.name)
    print('annotation', p.annotation)
    print('kind', p.kind)
    print('default', p.default)
```

A signature is metadata that describes the nature of a function—how you would call it, type hints, and so on. There are various things that you might do with a signature. One useful operation on signatures is comparison. For example, here's how you check to see if two functions have the same signature:

```
def func1(x, y):
    pass

def func2(x, y):
    pass

assert inspect.signature(func1) == inspect.signature(func2)
```

This kind of comparison might be useful in frameworks. For example, a framework could use signature comparison to see if you're writing functions or methods that conform to an expected prototype.

If stored in the `__signature__` attribute of a function, a signature will be shown in help messages and returned on further uses of `inspect.signature()`. For example:

```
def func(x, y, z=None):
    ...

func.__signature__ = inspect.signature(lambda x,y: None)
```

In this example, the optional argument `z` would be hidden in further inspection of `func`. Instead, the attached signature would be returned by `inspect.signature()`.

5.21 Environment Inspection

Functions can inspect their execution environment using the built-in functions `globals()` and `locals()`. `globals()` returns the dictionary that's serving as the global namespace. This is the same as the `func.__globals__` attribute. This is usually the same dictionary that's holding the contents of the enclosing module. `locals()` returns a dictionary containing the values of all local and closure variables. This dictionary is not the actual data structure used to hold these variables. Local variables can come from outer functions (via a closure) or be defined internally. `locals()` collects all of these variables and puts them into a dictionary for you. Changing an item in the `locals()` dictionary has no effect on the underlying variable. For example:

```
def func():
    y = 20
    locs = locals()
    locs['y'] = 30          # Try to change y
    print(locs['y'])        # Prints 30
    print(y)                # Prints 20
```

If you wanted a change to take effect, you'd have to copy it back into the local variable using normal assignment.

```
def func():
    y = 20
    locs = locals()
    locs['y'] = 30
    y = locs['y']
```

A function can obtain its own stack frame using `inspect.currentframe()`. A function can obtain the stack frame of its caller by following the stack trace through `f.f_back` attributes on the frame. Here is an example:

```
import inspect

def spam(x, y):
    z = x + y
    grok(z)

def grok(a):
    b = a * 10

    # outputs: {'a':5, 'b':50 }
    print(inspect.currentframe().f_locals)

    # outputs: {'x':2, 'y':3, 'z':5 }
    print(inspect.currentframe().f_back.f_locals)

spam(2, 3)
```

Sometimes you will see stack frames obtained using the `sys._getframe()` function instead. For example:

```
import sys
def grok(a):
    b = a * 10
    print(sys._getframe(0).f_locals)   # myself
    print(sys._getframe(1).f_locals)   # my caller
```

The attributes in Table 5.2 can be useful for inspecting frames.

Table 5.2 Frame Attributes

Attribute	Description
`f.f_back`	Previous stack frame (toward the caller)
`f.f_code`	Code object being executed
`f.f_locals`	Dictionary of local variables (`locals()`)
`f.f_globals`	Dictionary used for global variables (`globals()`)
`f.f_builtins`	Dictionary used for built-in names
`f.f_lineno`	Line number
`f.f_lasti`	Current instruction. This is an index into the bytecode string of `f_code`.
`f.f_trace`	Function called at start of each source code line

Looking at stack frames is useful for debugging and code inspection. For example, here's an interesting debug function that lets you view the values of the selected variables of the caller:

```
import inspect
from collections import ChainMap

def debug(*varnames):
    f = inspect.currentframe().f_back
    vars = ChainMap(f.f_locals, f.f_globals)
    print(f'{f.f_code.co_filename}:{f.f_lineno}')
    for name in varnames:
        print(f'    {name} = {vars[name]!r}')

# Example use
def func(x, y):
    z = x + y
    debug('x','y')  # Shows x and y along with file/line
    return z
```

5.22 Dynamic Code Execution and Creation

The `exec(str [, globals [, locals]])` function executes a string containing arbitrary Python code. The code supplied to `exec()` is executed as if the code actually appeared in place of the `exec` operation. Here's an example:

```
a = [3, 5, 10, 13]
exec('for i in a: print(i)')
```

The code given to `exec()` executes within the local and global namespace of the caller. However, be aware that changes to local variables have no effect. For example:

```
def func():
    x = 10
    exec("x = 20")
    print(x)         # Prints 10
```

The reasons for this have to do with the locals being a dictionary of collected local variables, not the actual local variables (see the previous section for more detail).

Optionally, `exec()` can accept one or two dictionary objects that serve as the global and local namespaces for the code to be executed, respectively. Here's an example:

```
globs = {'x': 7,
           'y': 10,
           'birds': ['Parrot', 'Swallow', 'Albatross']
          }

locs = { }

# Execute using the above dictionaries as the global and local namespace
exec('z = 3 * x + 4 * y', globs, locs)
exec('for b in birds: print(b)', globs, locs)
```

If you omit one or both namespaces, the current values of the global and local namespaces are used. If you only provide a dictionary for `globals`, it's used for both the globals and locals.

A common use of dynamic code execution is for creating functions and methods. For example, here's a function that creates an `__init__()` method for a class given a list of names:

```
def make_init(*names):
    parms = ','.join(names)
    code = f'def __init__(self, {parms}):\n'
    for name in names:
        code += f'    self.{name} = {name}\n'
    d = { }
    exec(code, d)
    return d['__init__']

# Example use
class Vector:
    __init__ = make_init('x','y','z')
```

This technique is used in various parts of the standard library. For example, `namedtuple()`, `@dataclass`, and similar features all rely on dynamic code creation with `exec()`.

5.23 Asynchronous Functions and await

Python provides a number of language features related to the asynchronous execution of code. These include so-called *async functions* (or coroutines) and *awaitables*. They are mostly used by programs involving concurrency and the `asyncio` module. However, other libraries may also build upon these.

An asynchronous function, or coroutine function, is defined by prefacing a normal function definition with the extra keyword `async`. For example:

```
async def greeting(name):
    print(f'Hello {name}')
```

If you call such a function, you'll find that it doesn't execute in the usual way—in fact, it doesn't execute at all. Instead, you get an instance of a coroutine object in return. For example:

```
>>> greeting('Guido')
<coroutine object greeting at 0x104176dc8>
>>>
```

To make the function run, it must execute under the supervision of other code. A common option is `asyncio`. For example:

```
>>> import asyncio
>>> asyncio.run(greeting('Guido'))
Hello Guido
>>>
```

This example brings up the most important feature of asynchronous functions—that they never execute on their own. Some kind of manager or library code is always required for their execution. It's not necessarily `asyncio` as shown, but something is always involved in making async functions run.

Aside from being managed, an asynchronous function evaluates in the same manner as any other Python function. Statements run in order and all of the usual control-flow features work. If you want to return a result, use the usual `return` statement. For example:

```
async def make_greeting(name):
    return f'Hello {name}'
```

The value given to `return` is returned by the outer `run()` function used to execute the async function. For example:

```
>>> import asyncio
>>> a = asyncio.run(make_greeting('Paula'))
>>> a
'Hello Paula'
>>>
```

Async functions can call other async functions using an `await` expression like this:

```
async def make_greeting(name):
    return f'Hello {name}'

async def main():
    for name in ['Paula', 'Thomas', 'Lewis']:
        a = await make_greeting(name)
        print(a)

# Run it.  Will see greetings for Paula, Thomas, and Lewis
asyncio.run(main())
```

Use of `await` is only valid within an enclosing `async` function definition. It's also a required part of making async functions execute. If you leave off the `await`, you'll find that the code breaks.

The requirement of using `await` hints at a general usage issue with asynchronous functions. Namely, their different evaluation model prevents them from being used in combination with other parts of Python. Specifically, it is never possible to write code that calls an async function from a non-async function:

```
async def twice(x):
    return 2 * x

def main():
    print(twice(2))        # Error. Doesn't execute the function.
    print(await twice(2))  # Error. Can't use await here.
```

Combining async and non-async functionality in the same application is a complex topic, especially if you consider some of the programming techniques involving higher-order functions, callbacks, and decorators. In most cases, support for asynchronous functions has to be built as a special case.

Python does precisely this for the iterator and context manager protocols. For example, an asynchronous context manager can be defined using `__aenter__()` and `__aexit__()` methods on a class like this:

```
class AsyncManager(object):
    def __init__(self, x):
        self.x = x

    async def yow(self):
        pass

    async def __aenter__(self):
        return self

    async def __aexit__(self, ty, val, tb):
        pass
```

Note that these methods are async functions and can thus execute other async functions using `await`. To use such a manager, you must use the special `async with` syntax that is only legal within an async function:

```
# Example use
async def main():
    async with AsyncManager(42) as m:
         await m.yow()

asyncio.run(main())
```

A class can similarly define an async iterator by defining methods `__aiter__()` and `__anext__()`. These are used by the `async for` statement which also may only appear inside an async function.

From a practical point of view, an async function behaves exactly the same as a normal function—it's just that it has to execute within a managed environment such as `asyncio`. Unless you've made a conscious decision to work in such an environment, you should move along and ignore async functions. You'll be a lot happier.

5.24 Final Words: Thoughts on Functions and Composition

Any system is built as a composition of components. In Python, these components include various sorts of libraries and objects. However, underlying everything are functions. Functions are the glue by which a system is put together and the basic mechanism of moving data around.

Much of the discussion in this chapter focused on the nature of functions and their interfaces. How are the inputs presented to a function? How are the outputs handled? How are errors reported? How can all of these things be more tightly controlled and better understood?

The interaction of functions as a potential source of complexity is worth thinking about when working on larger projects. It can often mean the difference between an intuitive easy-to-use API and a mess.

Index