Python Essentials

Modernize existing Python code and plan code migrations to Python using this definitive guide

Steven F. Lott
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 7 'Basic Function Definitions'
- A synopsis of the book’s content
- More information on Python Essentials
Steven F. Lott has been programming since the 70s, when computers were large, expensive, and rare. As a contract software developer and architect, he has worked on hundreds of projects, from very small to very large. He's been using Python to solve business problems for over 10 years.

He's particularly adept at struggling with thorny data representation problems.

He has also authored Mastering Object-oriented Python by Packt Publishing.

He is currently a technomad who lives in various places on the east coast of the US. His technology blog can be found at http://slott-softwarearchitect.blogspot.com.
Preface

Python programming should be expressive and elegant. In order for this to be true, the language itself must be easy to learn and easy to use. Any practical language—and its associated libraries—can present a daunting volume of information. In order to help someone learn Python, we've identified and described those features that seem essential.

Learning a language can be a long voyage. We'll pass numerous islands, archipelagos, inlets, and estuaries along the route. Our objective is to point out the key features that will be passed during the initial stages of this journey.

The concepts of data structures and algorithms are ever-present considerations in programming. Our overall approach is to introduce the various Python data structures first. As part of working with a given class of objects, the language statements are introduced later. One of Python's significant advantages over other languages is the rich collection of built-in data types. Selecting an appropriate representation of data can lead to elegant, high-performance applications.

An essential aspect of Python is its overall simplicity. There are very few operators and very few different kinds of statements. Much of the code we write can be generic with respect to the underlying data type. This allows us to easily exchange different data structure implementations as part of making tradeoffs between storage, performance, accuracy, and other considerations.

Some subject areas could take us well beyond the basics. Python's object-oriented programming features are rich enough to easily fill several large volumes. If we're also interested in functional programming features, we can study these in far more depth elsewhere. We'll touch only briefly on these subjects.
Preface

What this book covers

Chapter 1, Getting Started, addresses installing or upgrading Python. We explore Python's Read-Evaluate-Print Loop (REPL) as a way to interact with the language. We'll use this interactive Python mode as a way to explore most of the language features.

Chapter 2, Simple Data Types, introduces a few features concerning numbers and some simple collections. We'll look at Python's Unicode strings as well as byte strings, including some of the conversions between strings and numbers.

Chapter 3, Expressions and Output, provides more details on Python expression syntax and how the various numeric types relate to each other. We'll look at the coercion rules and the numeric tower. We'll look at the print() function, which is a common tool for looking at output.

Chapter 4, Variables, Assignment and Scoping Rules, shows how we assign names to objects. We look at a number of different assignment statements available in Python. We also explore the input() function, which parallels the print() function.

Chapter 5, Logic, Comparisons, and Conditions, shows the logical operators and literals that Python uses. We'll look at the comparison operators and how we use them. We'll look closely at the if statement.

Chapter 6, More Complex Data Types, shows the core features of the list, set, and dict built-in types. We use the for statement to work with these collections. We also use functions such as sum(), map(), and filter().

Chapter 7, Basic Function Definitions, introduces the syntax for the def statement as well as the return statement. Python offers a wide variety of ways to provide argument values to functions; we show a few of the alternatives.

Chapter 8, More Advanced Functions, extends the basic function definitions to include the yield statement. This allows us to write generator functions that will iterate over a sequence of data values. We look at a few functional programming features available via built-in functions as well as the modules in the Python Standard Library.

Chapter 9, Exceptions, shows how we handle and raise exceptions. This allows us to write programs which are considerably more flexible. A simple "happy path" can handle the bulk of the processing, and exception clauses can handle rare or unexpected alternative paths.
Chapter 10, *Files, Databases, Networks, and Contexts*, will introduce a number of features related to persistent storage. We'll look at Python's use of files and file-like objects. We'll also extend the concept of persistence to include some database features available in the *Python Standard Library*. This chapter will also include a review of the *with* statement for context management.

Chapter 11, *Class Definitions*, demonstrates the *class* statement and the essentials of object-oriented programming. We look at the basics of inheritance and how to define class-level (static) methods.

Chapter 12, *Scripts, Modules, Packages, Libraries, and Applications*, shows different ways in which we can create Python code files. We'll look at the formal structures of script, module, and package. We'll also look at informal concepts such as application, library, and framework.

Chapter 13, *Metaprogramming and Decorators*, introduces two concepts that can help us write Python code that manipulates Python code. Python makes metaprogramming relatively simple; we can leverage this to simplify certain types of programming where a common aspect doesn't fit neatly into a class hierarchy or library of functions.

Chapter 14, *Fit and Finish – Unit Testing, Packaging, and Documentation*, moves beyond the Python language into the idea of creating a complete, polished product. Any well-written program should include test cases and documentation. We show common ways to make sure this is done properly.

Chapter 15, *Next Steps*, will demonstrate four simple kinds of applications. We'll look at the command-line interface (CLI), graphic user interface (GUI), simple Web frameworks, as well as MapReduce applications.
Basic Function Definitions

Mathematically, a function is a mapping of values in a domain to values in a range. Functions like sine or cosine map values from a domain of angles to a range of real values between -1 and +1. The details of the mapping are summarized in the name, domain, and range. We'll use this function concept as a way to package our Python programming into something that allows us to summarize the implementation details using a name.

We'll look at how to define and evaluate Python functions. In this chapter, we'll focus on Python functions that simply return Python objects as the range of values. In Chapter 8, More Advanced Functions, we'll look at generator functions; these are iterators, which are used with a for loop to produce sequences of values.

Python functions offer optional parameters as well as a mixture of positional and keyword parameters. This allows us to define a single function which has a number of variant signatures, allowing considerable flexibility in how the function is used.

Looking at the five kinds of callables

Python offers five variations on the theme of a function. Each of these is a kind of callable object: we can call the object with argument values and it returns a result. Here's how we'll organize our exploration:

- Basic functions created with the def statement are the subject of this chapter.
- Lambda forms are a function definition reduced to parameters and an expression; this is also a topic within this chapter.
- Generator functions and the yield statement are something we'll look at in Chapter 8, More Advanced Functions. These functions are iterators which can provide multiple results.
Basic Function Definitions

- Function wrappers for class methods are something we'll look at in Chapter 11, Class Definitions. These are built-in functions which leverage features of a class. A function like `len()` is implemented by the `__len__()` method of a collection.

- Callable objects are also part of Chapter 11, Class Definitions. These are classes which include the `__call__()` method so that an instance of the class behaves like a basic function created with the `def` statement.

All of these are variations on a common theme. They are ways to package some functionality so that it has a name, input parameters, and a result. This allows us to decompose large, complex programs into smaller, easier-to-understand functions.

Defining functions with positional parameters

The essential Python function definition is built with the `def` statement. We provide a name, the names of the parameters, and an indented suite of statements that is the body of the function. The `return` statement provides the range of values.

The syntax looks like this:

```python
def prod(sequence):
    p = 1
    for item in sequence:
        p *= item
    return p
```

We've defined a name, `prod`, and provided a list of only one parameter, `sequence`. The body of the function includes three statements: assignment, `for`, and `return`. The expression in the `return` statement provides the resulting value.

This fits the mathematical idea of a function reasonably well. The domain of values is any numeric sequence, the range will be a value of the a type which reflects the data types in the sequence.

We evaluate a function by simply using the name and a specific value for the argument in an expression:

```python
>>> prod([1,2,3,4])
24
>>> prod(range(1,6))
120
```
In the first example, we provided a simple list display, \([1, 2, 3, 4]\), as an argument. This was assigned to the parameter of the function, `sequence`. The evaluation of the function returned the product of that sequential collection of items.

In the second example, we provided a `range()` object as the argument to the `prod()` function. This argument value is assigned to the parameter of the function. When used with a `for` loop, the range object behaves like a sequence collection, and a product is computed and returned.

**Defining multiple parameters**

Python offers us a variety of ways to assign values to parameters. In the simplest case, the argument values are assigned to the parameters based on position. Here's a function with two positional parameters:

```python
def main_sail_area(boom, mast):
    return (boom*mast)/1.8
```

We've defined a function that requires the length of the sail's boom, usually called the "E" dimension, and the height of the mast along which the sail is rigged, usually called the "P" dimension. Given these two numbers, and an assumption about the curvature of the sail, we return the approximate sail area.

We can evaluate this function providing the two positional parameters for boom length and mast height.

```python
>>> main_sail_area(15, 45)
375.0
```

We can define a function with any number of parameters. A function with a large number of parameters will tend to push the edge of the envelope on comprehensibility. A good function should have a tidy summary that makes it possible to understand the function's purpose without having to struggle with too many details.
**Using the return statement**

The `return` statement has two purposes: it ends the function's execution, and it can optionally provide the result value for the function. The `return` statement is optional. This leads to three use cases:

- **No return statement**: The function finishes at the end of the suite of statements. The return value is `None`.
- **A return statement with no expression**: The function finishes when the `return` statement is executed, the result is `None`.
- **A return statement with an expression**: The function finishes when the `return` statement is executed, the value of the expression is the result. A `return` statement with a list of expressions creates a *tuple*, suitable for multiple assignment.

Here's a function with no `return` statement:

```python
def boat_summary(name, rig, sails):
    print( "Boat {0}, {1} rig, {2:.0f} sq. ft.".format( 
        name, rig, sum(sails))
)
```

This function consists of a single expression statement that uses the `print()` function. There's no explicit `return` so the default return value will be `None`.

It's common to use a `return` statement to finish early when an exception condition has been met, otherwise you execute the rest of the suite of statements in the function definition. It looks like this:

```python
def mean_diff(data_sequence):
    s0, s1 = 0, 0
    for item in data_sequence:
        s0 += 1
        s1 += item
    if s0 < 2:
        return
    m= s1/s0
    for item in data_sequence:
        print(item, abs(item-m))
```
This function expects a collection of data. It will compute two values from that collection: \( s_0 \) and \( s_1 \). The \( s_0 \) value will be a count of items, the \( s_1 \) value will be the sum of the items. If the count is too small, the function simply returns. If the count is large enough, then additional processing is done: the values are printed along with the absolute difference between the value and the average.

There's no return statement at the end of the suite of statements, since this is not required. Using a return statement in the middle of a function allows us to avoid deeply-nested if statements.

Note that the variables \( s_0 \), \( s_1 \), and \( m \), are created in a local namespace that only exists while the function is being evaluated. Once the function is finished, the local namespace is removed, the reference counts are decremented and the interim objects are cleaned up. We'll look at additional details in the Working with namespaces section later in this chapter.

The built-in function \texttt{divmod()} \ returns two results. We often use multiple assignments like this: \( q, r = \text{divmod}(n, 16) \); it will assign the two results to two variables, \( q \) and \( r \). We can write a function that returns multiple values by including multiple expressions on the return statement.

In the Mutable and immutable argument values section, we'll show a function that has multiple return values.

### Evaluating a function with positional or keyword arguments

Python allows us to provide argument values with explicit parameter names. When we provide a name, it's called a keyword argument. For example, the \texttt{boat_summary()} \ function in the previous section can be used in a number of different ways.

We can provide the argument values positionally, like this:

```python
>>> sails = [358.3, 192.5, 379.75, 200.0]
>>> boat_summary("Red Ranger", "ketch", sails)
```

The arguments are assigned to the parameter variables of \texttt{name}, \texttt{rig}, and \texttt{sails} based on their position.

We can, as an alternative, do something like this:

```python
>>> boat_summary(sails=sails, rig="ketch", name="Red Ranger")
```
This example provides all three arguments with keywords. Note that the position doesn't matter when providing keyword arguments. The keyword arguments must be provided after any positional arguments, but the order among the keyword arguments doesn't matter, since they are assigned to parameters by name.

We can use a mixture of positional and keyword arguments. For this to work, Python uses two rules to map argument values to a function's parameters:

1. Match all positional arguments to parameters from left-to-right.
2. Match all keyword parameters by name.

There are several additional rules to handle duplicates and default values—which include optional parameters—described later, in the section called Defining optional parameters via default values.

In order for these rules to work properly, we must provide all positional parameters first, and then we can provide any keyword parameters after the positional parameters. We can't provide two values for the same parameter via position as well as keyword. Nor, for that matter, can we provide a keyword twice.

Here's a good example and a bad example:

```python
>>> boat_summary("Red Ranger", sails=sails, rig="ketch")
```

```python
>>> boat_summary("Red Ranger", sails=sails, rig="ketch", name="Red Ranger")
```

In the first example, the `name` parameter is matched positionally. The `sails` and `rig` parameters were matched by keyword.

In the second example, there are both positional and keyword values for the `name` variable. This will raise a `TypeError` exception.

Because of this, it's very important to choose parameter variable names wisely. A good choice of parameter name can make keyword argument function evaluation very clear.

**Writing a function's docstring**

In order to save space, we haven't provided many examples of functions with docstrings. We'll address docstrings in detail in Chapter 14, Fit and Finish – Unit Testing, Packaging, and Documentation. For now, we need to be aware that every function should, at the very least, have a summary. The summary is included as a triple-quoted string that must be the very first expression in the suite of statements of the function.
A function with a docstring looks like this:

```python
def jib(foot, height):
    ""
    jib(foot, height) -> area of given jib sail.
    ""
    return (foot*height)/2
```

This particular triple-quoted string serves two purposes. First, it summarizes what the function does. We can read this when we look at the source file. We can also see this when we use `help(jib)`.

The second purpose for this docstring is a way to provide a concrete example of how the function is used. The examples always look like they are simply copied from a REPL session and pasted into the docstring comment.

These REPL-formatted examples are located by using the `doctest` tool. After locating the examples, this tool can run the code to confirm that it works as advertised. All of the examples in this book were tested using `doctest`. While the details of testing are part of Chapter 14, Fit and Finish – Unit Testing, Packaging, and Documentation, it’s important to consider writing docstrings in every function.

## Mutable and immutable argument values

In some programming languages, there are multiple function evaluation strategies, including call-by-value and call-by-reference. In call-by-value semantics, copies of argument values are assigned to the parameter variables in a function. In call-by-reference semantics, a reference to a variable is used in the function. This means that an assignment statement inside a function could replace the value of a variable outside the function. Neither of these types of semantics apply to Python.

Python uses a mechanism named "call-by-sharing" or "call-by-object". A function is given a reference to the original object. If that object is mutable, the function can mutate the object. The function cannot, however, assign to variables outside the function via the parameter variables. The function shares the objects, not the variables to which the objects are assigned.

One of the most important consequences is that the body of a function can assign new values to parameter variables without having any impact on the original arguments which are passed to a function. The parameter variables are strictly local to the function.
Basic Function Definitions

Here's a function that assigns new values to the parameter variable:

```python
def get_data(input_string):
    input_string= input_string.strip()
    input_string, _, _ = input_string.partition("#")
    input_string= input_string.rstrip()
    name, _, value = input_string.partition('=')
    return name, value
```

This function evaluates the `strip()` method of the `input_string` variable and assigns the resulting string to the parameter variable. It applies the `partition()` method to the new value of the `input_string` variable and assigns one of the three resulting strings to the parameter variable. It then returns this string object, assigning it to the parameter variable yet again.

None of the assignment statements to the `input_string` parameter variable have any effect on any variables outside the function. When a function is evaluated, a separate namespace is used for the parameters and other local variables.

Another consequence of the way Python works is that when we provide mutable objects as arguments, these objects can be updated by methods evaluated inside a function. The function's parameter variables will be references to the original mutable objects, and we can evaluate methods like the `remove()` or `pop()` functions that change the referenced object.

Here's a function which updates a list argument by removing selected values:

```python
def remove_mod(some_list, modulus):
    for item in some_list[:]
        if item % modulus == 0:
            some_list.remove(item)
```

This function expects a mutable object like a list, named `some_list`, and a value, named `modulus`. The function makes a temporary copy of the argument value using `some_list[:]`. For each value in this copy that is a multiple of the `modulus` value, we'll remove that copy from the original `some_list` object. This will mutate the original object.

When we evaluate this function, it looks like this:

```python
>>> data= list(range(10))
>>> remove_mod(data, 5)
>>> remove_mod(data, 7)
>>> data
[1, 2, 3, 4, 6, 8, 9]
```
We've created a simple list and assigned it to the data variable. This object referred to by the data variable was mutated by the remove_mod() function. All multiples of five and seven were discarded from the sequence.

In this function, we need to create a temporary copy of the input list object before we can start removing values. If we try to iterate through a list while simultaneously removing items from that list, we'll get results that don't appear correct. It helps to have the original values separate from the mutating list.

A function can create variables in the global namespace, and other non-local namespaces, by making special arrangements. This is done with the global and nonlocal statements shown in the Working with namespaces section.

**Defining optional parameters via default values**

Python lets us provide a default value for a parameter. Parameters with default values are optional. The standard library is full of functions with optional parameters. One example is the int() function. We can use int("48897") to convert a string to an integer, assuming that the string represents a number in base 10. We can use int("48897", 16) to explicitly state that the string should be treated as a hexadecimal value. The default value for the base parameter is 10.

Remember that we can use keyword arguments for a function. This means that we might want to write something like this: int("48897", base=16), to make it abundantly clear what the second argument to the int() function is being used for.

Earlier, we listed two rules for matching argument values to parameters. When we introduce default values, we add two more rules.

1. Match all positional arguments to parameters from left-to-right.
2. Match all keyword parameters. In case of already-assigned positional parameters, raise a TypeError exception.
3. Set default values for any missing parameters.
4. In case there are parameters with no values, raise a TypeError exception.

Note: This is not the final set of rules; there are a few more features to cover.
One important consequence of these rules is that the required parameters—those without default values—must be defined first. Parameters with default values must be defined last. The "required parameters first, optional parameters last" rule assures us that the positional matching process works.

We provide the default value in the function definition. Here's an example:

```python
import random
def dice(n=2, sides=6):
    return [random.randint(1,sides) for i in range(n)]
```

We've imported the `random` module so that we can use the `random.randint()` function. Our `dice()` function has two parameters, both of which have default values. The `n` parameter, if not provided, will have a value of 2. The `sides` parameter, if omitted, will have a value of 6.

The body of this function is a list comprehension: it uses a generator expression to build a list of individual values. We'll look at generator expressions in detail in Chapter 8, More Advanced Functions. For now, we can observe that it uses the `random.randint(1,sides)` function to generate numbers between 1 and the value of the `sides` parameter. The comprehension includes a `for` clause that iterates through `n` values.

We can use this function in a number of different ways. Here are some examples:

```python
>>> dice()
[6, 6]
>>> dice(6)
[3, 6, 2, 2, 1, 5]
>>> dice(4, sides=4)
[3, 3, 4, 3]
```

The first example relies on default values to emulate the pair of dice commonly used in casino games like Craps. The second example uses six dice, typical for games like 10,000 (sometimes called Zilch or Crap Out.). The third example uses four four-sided dice, typical of games that use a variety of polyhedral dice.

A note on testing: in order to provide repeatable unit tests for functions that involve the `random` module, we've set a specific seed value using `random.seed("test")`. 

---

[130]
A warning about mutable default values

Here's a pathological example. This shows a very bad programming practice; it's a mistake that many Python programmers make when they first start working with default values.

This is a very bad idea:

```python
def more_dice(n, collection=[]):
    for i in range(n):
        collection.append(random.randint(1,6))
    return collection
```

We've defined a simple function with only two parameter variables, `n` and `collection`. The collection has a default value of an empty list. (Spoiler alert: this will turn out to be a mistake.) The function will append a number of simulated six-sided dice to the given collection.

The function returns a value as well as mutating an argument. This means that we'll see the `return` value printed when we use this function in the REPL.

We can use this for games like Yacht, also called Generala or Poker Dice. A player has a "hand" of dice from which we'll remove dice and append new dice rolls.

One use case is to create a list object and use this as an argument to the `more_dice()` function. This list object would get updated nicely. Here's how that would work:

```python
>>> hand1= []
>>> more_dice(5, hand1)
[6, 6, 3, 6, 2]
>>> hand1
[6, 6, 3, 6, 2]
```

We've created an empty list and assigned it to the `hand` variable. We provided this sequence object to the `more_dice()` function to have five values appended to the `hand` object. This gave us an initial roll of three sixes, a three, and a two. We can remove the two and three from the `hand1` object; we can reuse it with `more_dice(2, hand1)` to put two more dice into the hand.
We can use another empty sequence as an argument to deal a second hand. Except for the results, it's otherwise identical to the first example:

```python
>>> hand2= []
>>> more_dice(5, hand2)
[5, 4, 2, 2, 5]
>>> hand2
[5, 4, 2, 2, 5]
```

Everything seems to work properly. This is because we're providing an explicit argument for the collection parameter. Each of the hand objects is a distinct, empty list. Let's try to use the default value for the `collection` parameter.

In this third example, we won't provide an argument, but rely on the default sequence returned by the `more_dice()` function:

```python
>>> hand1= more_dice(5)
>>> hand1
[6, 6, 3, 6, 2]
>>> hand2= more_dice(5)
>>> hand2
[6, 6, 6, 2, 1, 5, 4, 2, 2, 5]
```

Wait. What just happened? How is this possible?

As a hint, we'll need to search through the code for an object with a hidden, shared state. Earlier, we noted that a default list object would be a problem. This hidden list object is getting re-used.

What happens is this:

1. When the `def` statement is executed, the expressions that define the parameter defaults are evaluated. This means that a single mutable list object is created as the default object for the `collection` parameter.
2. When the `more_dice()` function is evaluated without an argument for the `collection` parameter, the one-and-only mutable list object is used as the default object. What's important is that a single mutable object is being reused. If, at any point, we update this object, that mutation applies to all shared uses of the object. Since it's returned by the function, this single list could be assigned to several variables.
3. When the `more_dice()` function is evaluated a second time without an argument for the `collection` parameter, the mutated list object is re-used as the default.
From this, we can see that a mutable object is a terrible choice of a default value.

Generally, we have to do something like this:

```python
def more_dice_good(n, collection=None):
    if collection is None:
        collection = []
    for i in range(n):
        collection.append(random.randint(1,6))
    return collection
```

This function uses an immutable and easily-recognized default value of `None`. If no argument value is provided for the `collection` variable, it will be set to `None`. We can replace the `None` value with a brand new `list` object created when the function is evaluated. We can then update this new `list` object, confident that we're not corrupting any mutable default object which is being reused.

Don't use a mutable object as a default value for a parameter.
Avoid `list`, `dict`, `set`, and any other mutable type, as default parameter values. Use `None` as a default; replace the `None` with a new, empty mutable object.
You've been warned.

This can lead to mistakes. It's a consequence of the way that function definition works and call-by-sharing semantics.

It's possible to exploit this intentionally: we can use a mutable default value as a cache to retain values, creating functions which have hysteresis. A callable object may be a better way to implement functions with an internal cache or buffer. See Chapter 11, Class Definitions, for more information.

### Using the "everything else" notations of * and **

Python offers even more flexibility in how we can define positional and keyword parameters for a function. The examples we've seen are all limited to a fixed and finite collection of argument values. Python allows us to write functions that have an essentially unlimited number of positional as well as keyword argument values.
Python will create a *tuple* of all unmatched positional parameters. It will also create a dictionary of all unmatched keyword parameters. This allows us to write functions that can be used like this:

```python
>>> prod2(1, 2, 3, 4)
24
```

This function accepts an arbitrary number of positional arguments. Compare this with the `prod()` function shown previously. Our previous example required a single sequence object, and we had to use that function as follows:

```python
>>> prod([1, 2, 3, 4])
24
```

The `prod2()` function will create a product of all argument values. Since the `prod2()` function can work with an unlimited collection of positional arguments, this leads to slightly simpler syntax for this function.

In order to write a function with an unlimited number of positional arguments, we must provide one parameter which has a * prefix. It looks like this:

```python
def prod2(*args):
    p = 1
    for item in args:
        p *= item
    return p
```

The definition of `prod2()` has all positional arguments assigned to the * prefix parameter, `*args`. The value of the `args` parameter is a tuple of the argument values.

Here's a function which uses a mixture of positional and keyword parameters:

```python
def boat_summary2(name, rig, **sails):
    print("Boat {0}, {1} rig, {2:.0f} sq. ft.\".format(
        name, rig, sum(sails.values())))
```

This function will accept two arguments, `name` and `rig`. These can be supplied by position or by keyword. Any additional keyword parameters—other than `name` and `rig`—are collected into a dictionary and assigned to the `sails` parameter. The `sails.values()` expression extracts just the values from the `sails` dictionary; these are added together to write the final summary line.

Here's one of many ways that we can use this function:

```python
>>> boat_summary2("Red Ranger", rig="ketch",
...     main=358.3, mizzen=192.5, yankee=379.75, staysl=200 )
```
We've provided the first argument value by position; this will be assigned to the first positional parameter, name. We've provided one of the defined parameters using a keyword argument, rig. The remaining keyword arguments are collected into a dictionary and assigned to the parameter named sails.

The sails dictionary will be assigned a value similar to this:

```python
{'main': 358.3, 'mizzen': 192.5, 'yankee': 379.75, 'staysl': 200}
```

We can use any dictionary processing on this mapping since it's a proper dict object.

Earlier, we provided four rules for matching argument values with parameters. Here is a more complete set of rules for matching argument values with function parameters:

1. Match all positional arguments to parameters from left-to-right.
2. If there are more positional arguments than parameter names:
   1. If there's a parameter name with a * prefix, assign a tuple of remaining values to the prefixed parameter.
   2. If there's no parameter with the * prefix, raise a TypeError exception.
3. Match all keyword parameters. In case of already-assigned positional parameters, raise a TypeError exception.
4. If there are more keyword arguments than parameter names:
   1. If there's a parameter name with a ** prefix, assign dict of remaining keywords and values to the prefixed parameter.
   2. If there's no parameter with the ** prefix, raise a TypeError exception.
5. Apply default values to missing parameters.
6. In case of parameters which still have no values, raise a TypeError exception.

A consequence of these rules is that, at most, only one parameter can have a * prefix; similarly, at most, only one parameter can have a ** prefix. These special cases must be given after all of the other parameters. The * prefix variable will be assigned an empty tuple if there are no surplus positional parameters. The ** prefix variable will be assigned an empty dictionary if there no surplus keyword parameters.

When invoking a function, we must provide positional argument values first. We can provide keyword argument values in any order.
Basic Function Definitions

Using sequences and dictionaries to fill in *args and *kw

The prod2() function, shown earlier, expects individual values which are collected into a single *args tuple. If we invoke the function with prod2 (1, 2, 3, 4, 5), then a tuple built from the five positional parameters is assigned to the args parameter.

What if we want to provide a list to the prod2() function? How can we, in effect, write prod2(some_list[0], some_list[1], some_list[2], …)?

When we call a function using prod2(*some_sequence), then the values of the given argument sequence are matched to positional parameters. Item zero from the argument sequence becomes the first positional parameter. Item one from the sequence becomes the second parameter, and so on. Each item is assigned until they’re all used up. If there are extra argument values, and the function was defined with a parameter using a * prefix, the extra argument values are assigned to the * prefix parameter.

Because of this, we can use prod2(*range(1, 10)) easily. This is effectively prod2(1, 2, 3, 4, 5, …, 9). Since all of the positional argument values are assigned to the *-prefix args variable, we can use this function with individual values, like this: prod2(1, 2, 3, 4). We can provide a sequence of values, like this: prod2(*sequence).

We have a similar technique for providing a dictionary of keyword arguments to a function. We can do this:

```python
>>> rr_args = dict(
...    name="Red Ranger", rig="ketch",
...    main=358.3, mizzen=192.5, yankee=379.75, staysl=200
... )
>>> boat_summary2(**rr_args)
Boat Red Ranger, ketch rig, 1131 sq. ft.
```

We’ve created a dictionary with all of the arguments defined via their keywords. This uses a handy feature of the dict() function where all of the keyword arguments are used to build a dictionary object. We assigned that dictionary to the rr_args variable. When we invoked the boat_summary2() function, we used the **rr_args parameter to force each key and value in the rr_args dictionary to be matched against parameters to the function. This means that the values associated with the name and rig keys in the dictionary will be matched with the name and rig parameters. All other keys in the dictionary will be assigned the sails parameter.
These techniques allow us to build function arguments dynamically. This gives us tremendous flexibility in how we define and use Python functions.

**Nested function definitions**

We can include anything inside a function definition, even another function definition. When we look at decorators in *Chapter 13, Metaprogramming and Decorators*, we'll see cases of a function which includes a nested function definition.

We can include `import` statements within a function definition. An `import` statement is only really executed once. There's a global collection of imported modules. The name, however, would be localized to the function doing the import.

The general advice is given in the *Zen of Python* poem by Tim Peters:

> Flat is better than nested.

We'll generally strive to have functions defined in a relatively simple, flat sequence. We'll avoid nesting unless it's truly required, as it is when creating decorators.

**Working with namespaces**

When a function is evaluated, Python creates a local namespace. The parameter variables are created in this local namespace when the argument values (or default values) are assigned. Any variables that are created in the suite of statements in the function's body are also created in this local namespace.

As we noted in *Chapter 4, Variables, Assignment and Scoping Rules*, each object has a reference counter. An object provided as an argument to a function will have the reference count incremented during the execution of the function's suite of statements.

When the function finishes—either because of an explicit `return` statement or the implicit return at the end of the suite—the namespace is removed. This will decrement the number of references to the argument objects.

When we evaluate an expression like `more_dice_good(2, hand)`, the literal integer 2 will be assigned to the `n` parameter variable. Its reference count will be one during the execution of the function. The object assigned to the `hand` variable will be assigned to the `collection` parameter. This object will have a reference count of two during the execution of the function.
Basic Function Definitions

When the function exits, the namespace is removed, which removes the two parameter variables. The literal 2 object, assigned to the `n` variable, will wind up with a reference count of zero, and that `int` object can be removed from memory. The object assigned to the `collection` variable will have its reference count decreased from two to one; it will not be removed from memory. This object will still be assigned to the `hand` variable, and can continue to be used elsewhere.

This use of a local namespace allows us to freely assign objects to parameters without having the objects overwritten or removed from memory. It also allows us to freely create intermediate variables within the body of a function, secure in the knowledge that the variable will not overwrite some other variable used elsewhere in the script.

When we reference a variable, Python looks in two places for the variable. It looks first in the local namespace. If the variable isn't found, Python then searches the global namespace.

When we import a module, like `random`, we generally write `import` at the beginning of our script so that the module is imported into the global namespace. This means that a function which uses `random.randint()` will first check the local namespace for `random`; failing to find that, it will check the global namespace and find the imported module.

This fallback to the global namespace allows us to reuse imported modules, function definitions, and class definitions freely within a script file. We can also—to an extent—share global variables. The default behavior is that we can read the values of global variables, but we can't easily update them.

If we write `global_variable = global_variable + 1` in a function, we can fetch the value of a global variable named `global_variable`. The assignment, however, will create a new variable in the local namespace with the name `global_variable`. The actual global variable will remain untouched.

**Assigning a global variable**

What if we want to assign values to a variable which has not been provided as an argument? We can write a function which will update global variables. This can lead to confusing programs because several functions may share common states via the global variable.
To create names in the global namespace instead of a local namespace, we use the `global` statement. This identifies the variables which must be found in the global namespace instead of the local namespace. Here's a function which updates a global variable:

```python
import random

def roll_dice_count_7():
    global sevens
    d = random.randint(1, 6), random.randint(1, 6)
    if d[0] + d[1] == 7:
        sevens += 1
    return d
```

We've defined a function and used the `global` statement to state that the variable named `sevens` will be found in the global namespace. We've created two random numbers, and assigned the pair to a local variable, `d`. This variable will be created in the local namespace, and won't conflict with any other variables defined in other namespaces.

Each time the pair of dice total seven, the global variable is updated. This is a side effect that can be confusing. It must be documented explicitly, and it requires some careful unit testing.

The two built-in functions, `globals()` and `locals()`, can help clarify the variables available when this function is being evaluated. If we add a `print()` function right before the `return` statement, we'll see results (with some details elided) like this:

```python
globals={'__cached__': None,
'__loader__': <_frozen_importlib.SourceFileLoader object at 0x100623750>,
'sevens': 20,
'__name__': '__main__',
'__file__': '…',
… etc.
'roll_dice_count_7': <function roll_dice_count_7 at 0x10216e710>,
'random': <module 'random' from '…'>}
locals={'d': (2, 1)}
```

The `globals` function includes variables like `sevens`, it includes the random module, and the `roll_dice_count_7` function. It includes some system variables: `__cached__`, `__loader__`, `__name__`, and `__file__`.

The `locals` function includes the local variable `d` and nothing more.
Assigning a non-local variable

When one function is defined inside another function, the outer function can contain variables which are neither local to the inner function nor global. We call these non-local variables. There are situations where we might want to set a variable which is part of an enclosing function.

Nested function definitions are most commonly used when defining decorators. We'll look at this in Chapter 13, Metaprogramming and Decorators.

Here's a contrived example of nested functions and a non-local shared variable:

```python
def roll_nl(n=2, d=6):
    def dice():
        nonlocal total
        points= tuple(random.randint(1,d) for _ in range(n))
        total = sum(points)
        return points
    total= 0
    dice()
    return total
```

We've defined a function, roll_nl(), which will simulate rolls of dice. The function's body includes a nested function definition, dice(). The rest of the body creates the variable total, evaluates the internal dice() function, and returns the value of the total variable.

How did the total variable get set to anything other than zero? It isn't updated in the body of the roll_nl() function.

Within the nested dice() function, there's a nonlocal reference to a variable named total. This variable must exist in an outer namespace, but not necessarily the global namespace. The dice() function creates a tuple object with the values of n dice. This expression builds a tuple from the result of a generator function. It updates the nonlocal total variable the sum of the points tuple. The nonlocal statement assures us that the total variable is part of the container for the dice() function. The return value of the dice() function is the tuple of dice, a value this isn't really used.
Defining lambdas

A lambda form is a degenerate kind of function. A lambda doesn't even have a name: it has only parameters and a single expression. We create a lambda by providing the parameter names and the expression. It looks like this:

\[
\text{lambda } x: x[0]+x[1]
\]

This kind of thing is helpful in the context of Python's higher-order functions. We often use lambdas with `max()`, `min()`, `sorted()`, `map()`, `filter()`, or `list.sort()`. Here's a simple example:

```python
>>> colors = [(255, 160, 137),
            (143, 80, 157),
            (255, 255, 255),
            (162, 173, 208),
            (255, 67, 164),
            ]

>>> sorted(colors)
[(143, 80, 157), (255, 160, 137), (255, 255, 255)]

>>> sorted(colors,
          key= lambda rgb: (rgb[0]+rgb[1]+rgb[2])/3)
[(143, 80, 157), (255, 160, 137), (255, 255, 255)]
```

We've created a simple list object which has four RGB color values. If we use the `sorted()` function on this list, the colors are sorted into order by the red component value. If the red components are equal, then the green component is used. In the rare case that the red and green components are equal the blue component is used.

If we want colors sorted by brightness, we can't simply sort by red, green, and blue. The perception of brightness is subtle and a number of formulae approximate the phenomena. We've picked just one, which is to average the RGB values. This formula doesn't take into account the fact that our eyes are more sensitive to green.

The `sorted()` function accepts a second parameter, `key`, which we've provided as a keyword argument in the second example. Rather than write a complete function definition that would only really embody a single expression, we've packaged the expression, \((rgb[0]+rgb[1]+rgb[2])/3\), as a lambda.
Basic Function Definitions

The syntax `lambda rgb: (rgb[0]+rgb[1]+rgb[2])/3` is equivalent to the following function definition.

```python
def brightness(rgb):
    return (rgb[0]+rgb[1]+rgb[2])/3
```

The lambda is more compact. If we only need this expression in one place, a reusable function may not be appropriate. A lambda is an easy way to provide a simple expression with minimal overhead. If we think we need to write complex lambdas—more than a simple expression—or we need to reuse a lambda, then we should consider using a proper function.

Writing additional function annotations

The Python Enhancement Proposal (PEP) number 3107 specifies additional annotations which can be applied to a function definition. Additionally, PEPs 482, 483, and 484 cover some related ideas.

This is important only because Python has some optional syntax that we may see. In Python 3.5, there may be additional tools for the type of information provided in this optional syntax. The annotated code can look like this:

```python
def roller( n: int, d: int = 6 ) -> tuple:
    return tuple(random.randint(1,d) for _ in range(n))
```

This function includes additional `: expression` annotations after each parameter. It also includes a `-> expression` annotation to show the return type of the function. All of the annotation expressions in this example are the names of built-in types.

In order to describe more complex structures, an additional typing module can offer the tools for defining a more exact `Tuple[int, ...]` as the return type for this function. This is an exciting development that may avoid certain kinds of bugs.

These annotations are legal Python3 syntax. They have no formally-defined semantics, which means that they are optional. There are some enhancement projects that are working on leveraging these optional annotations and creating tools that can use the information provided there. It's rarely used, but perfectly legal.
Summary

We've looked at a number of Python features for function definition. We've looked at how we define the name, and the parameters to a function, providing default values to make parameters optional. We've also looked at how we can provide arguments to a function: we can provide arguments by position, or by using the parameter variable name as a keyword. We can evaluate `function(*args)` to map a sequence of values to parameters by position. We can also evaluate `function(**kw)` to map a dictionary of values to parameters by name. And, of course, we can combine these two techniques.

We've looked at how functions return values via the `return` statement. We've also looked at functions which don't return a value. Technically, they return a value of `None`, which the rest of the Python programming ignores.

We've looked at the all-important issue of attempting to use a mutable object as a default value in a function definition. Most of the time, a mutable object as a default is going to create problems.

In addition to the basics of function definition, we looked at how local variables are assigned to temporary namespaces. We also looked at how we can use the `global` statement to create variables in the global namespace. We also looked at how a nested function definition can manipulate variables that are nonlocal to the nested function, but not properly global to the container function.

In Chapter 8, More Advanced Functions, we'll look at generator expressions and functions. These are functions which are iterable, and work cooperatively with the `for` loop to handle collections and sequences of data.
Where to buy this book
You can buy Python Essentials from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.