

00001100

CHAPTER 12

Modules and Functions

Ch-12: Modules and Functions

Learning Objectives

- *State the purpose and use of modules and functions*
- *State the naming conventions used to name modules and functions*
- *Explain what it means to minimize coupling and maximize cohesion*
- *Define the terms parameter and argument*
- *Use the keyword `def` to define functions*
- *State the purpose of function parameters*
- *Explain the scoping rules for function parameters*
- *Pass arguments to functions*
- *Return data from functions*
- *Explain why functions are considered first class objects*
- *Pass functions as arguments to functions*
- *Use docstrings to document code*
- *Use keyword parameters in function definitions*
- *Define default argument values*
- *Employ special parameters `*args` and `**kwargs` in function definitions*

0
0
0
0
1
1
0
0

INTRODUCTION

Modules and functions form the bedrock of reusable code. If you’ve read chapters 1-11, then you’re already acquainted with modules and functions, as well as methods, functions’ object-oriented cousins. It’s time now to dive deeper into how to effectively employ modules and functions in your programs.

In this chapter, you’ll learn the purpose and use of modules and functions. I’m treating these two topics together in one chapter because both are used to organize your source code. With regards to modules, you need to know what they are, how to name them, and how to use them to impose a first-order architectural structure upon your applications.

With regards to functions, you’ll learn how to define them, how to name them, how to declare function parameters, how to pass arguments to functions, how to return data from functions, and learn why functions are considered first class objects in Python.

Along the way, you’ll learn why it takes more than just gathering up code willy-nilly and giving it a name to create a useful function. A function must have a singular purpose; it must implement and perform one well-defined task and perform that task without causing unknown side effects in other parts of the program. In more formal terms, a function must *maximize cohesion* and *minimize coupling*. The concepts of maximal cohesion and minimal coupling apply to modules as well. Together, modules and functions facilitate structured programming.

Many of the concepts I discuss in this chapter trace their origins back to the software crisis of the 1960s when programmers, grappling with ever-increasing software complexity, began to formalize and codify software design and coding practices in an effort to write more *reliable*, *reusable*, and *maintainable* software. The concepts of maximizing module cohesion and minimizing module coupling have been passed down to us from this era and are as applicable today as they were then.

Everything you learn about functions in this chapter applies as well to methods. You’ll learn more about classes, methods, and object-oriented programming in Part IV: Object-Oriented Programming.

1 MODULES

Modules are files that contain Python source code. By convention, a Python source code file ends with the suffix “.py”. When Python reads a source file and loads it into the interpreter, the filename becomes the module name. More specifically, the module’s `__name__` attribute is set to the module’s filename sans the .py suffix, unless the module is directly executed by the Python interpreter and serves as the entry point to the application, in which case its `__name__` attribute is set to “`__main__`”. For a more complete discussion of this process refer back to Chapter 5, section “How Python Runs Programs” on page 202.

Modules serve as the first line of application code organization or *architecture*. An application *architecture* is a conceptual and physical assignment of responsibilities, features, and services to application components. A thoughtfully designed application architecture is easier to comprehend and maintain than one poorly designed. Simple Python scripts may contain only one module, while more complex applications will consist of multiple modules. Going forward, you need to know *how* to name modules as well as *what* to name them.

1.1 MODULE NAMING

The PEP 8 - Style Guide for Python Code is the goto reference for Python naming conventions and other source code formatting considerations. I will refer to it simply as PEP 8. Regarding module names, PEP 8 recommends using short, all lower-case names. You may use underscores in a module name to improve readability.

1.1.1 VALID MODULE NAMES

When you name something in a Python program, be it a module, function, variable, class, or other object, you are creating an *identifier* formed from of a set of characters recognized as valid in the programming language. Python defines these in [Section 2.3 of the Lexical Analysis section of the Python documentation](#), with modifications and additions detailed in [PEP 3131 — Supporting Non-ASCII Identifiers](#).

Here’s the short story: Valid identifier names include the upper-case characters A–Z, lower-case characters a–z, numbers 0–9, and underscores ‘_’. A module name *can* start with an upper or lower-case character, followed by upper or lower-case characters, numbers, or underscores, however, by convention, and as discussed in PEP-8, **use only lower-case characters, keep module names short, and add underscores as necessary** to make the module name easier for mere mortals to decipher. Table 12-1 offers a few examples of good and not-so-good module names.

Module Name	Comments
computersimulator.py	Good, but computer_simulator.py would be even better. Use an underscore to separate words in multi-word module names.
utils.py	OK, but vague.
disk_utils.py	Much better.
classes.py	Nope! The module name provides no hint regarding the module’s intended purpose. Thinking in object-oriented terms, a module with multiple classes is bad juju. However, even the assumption that the module contains class definitions may be all wrong.
my_first_program.py	It’s a valid module name and one a lot of beginners use when first learning to code, but offers little to suggest what it actually does.
hello_world.py	Another beginner module name but this time a bit more descriptive.
Person.py	OK, but favor the use of lower-case characters: person.py From an object-oriented point of view, a module named person.py would contain the definition of a class named Person. You’ll learn more about module naming for object-oriented programming in Part IV, Chapter 17: Classes.
l89g5.py	Nope! While a legal module name, it’s awful because it offers no hint as to what it contains and forces developers to explore the file to make sense of it.

Table 12-1: Good and Not-So-Good Module Names

1.1.2 CHOOSING APPROPRIATE MODULE NAMES

OK, now that you know *how* to name a module, you need to put some thought into *what* to name a module. Again, PEP 8 offers some guidance. It recommends you name modules according to their function, not their implementation. What does this mean, exactly?

This is actually easier than it sounds, and in my opinion, beginners struggle with module naming mostly because they are not exposed early on to complex application architectures. In other words, unless you work on an application that can be logically organized along distinct boundaries of responsibility, you will generally tend to throw all your source code into one all-encompassing module and be done with it.

Take for example an application that requires some sort of interaction with a user. This interaction might be via a console or graphical user interface. This same application may also process data entered by the user and store it in a database. From the viewpoint of *responsibilities*, the application can be considered to consist of three: *user interface*, *application logic*, and data storage or *database logic*. A naive implementation of such an application may group all these responsibilities into one monolithic module as shown in figure 12-1.

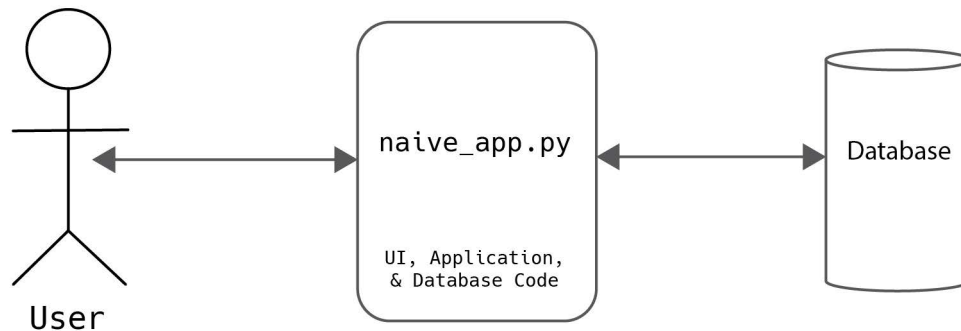


Figure 12-1: All Responsibilities Located In Single Monolithic Module

Referring to figure 12-1 — Single-module applications are not bad per se, but they do become problematic as application complexity increases. For example, if the naive application expands in scope, it will prove helpful to structure the application into multiple modules whose names reflect their responsibilities as shown in figure 12-2.

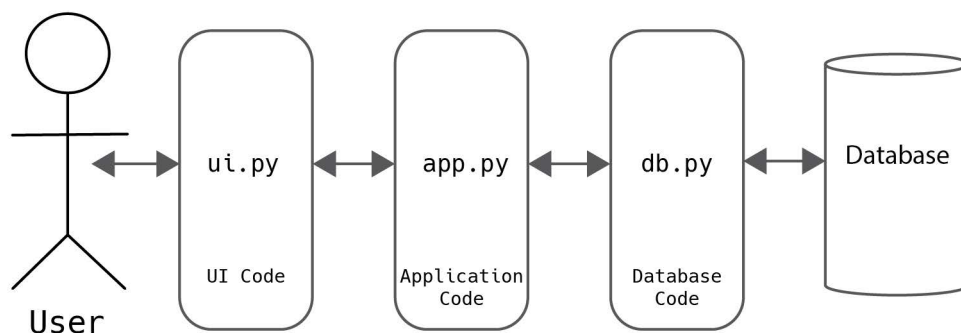


Figure 12-2: Application Organized into Separate Modules Based on Responsibility

Referring to figure 12-2 — This is a step in the right direction. Modules that focus on a single responsibility tend to contain less code and are easier to comprehend. Understanding what a module does or is intended to do goes a long way towards making a module easier to maintain.

Depending on your requirements regarding code extensibility, maintainability, and reliability, this multi-tiered application architecture may suffice as-is or be just the first iteration of many before settling on a final design.

1.2 ADDING AN EXPLICIT ENTRY POINT MODULE

You should know by now that when you run a module directly with the `python` command its `__name__` property is set to “`__main__`”, however, with multi-module applications, it’s not always clear nor intuitive which module is intended to serve as the entry point. That’s why, for multi-module applications, I like to add a module named `main.py` to serve as the explicit entry point for the application as shown in figure 12-3.

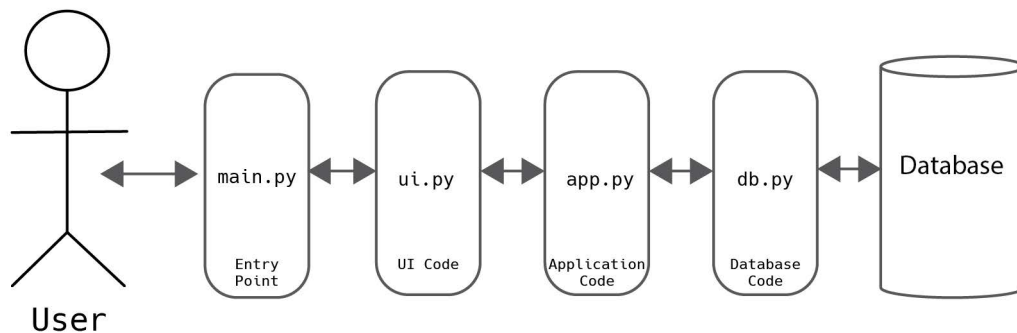


Figure 12-3: Adding a `main.py` Module to Serve as the Explicit Entry Point

Referring to figure 12-3 — The `main.py` module clearly indicates it’s the module that should be run by the Python interpreter. It clarifies your application design and eliminates guessing on the part of application users.

QUICK REVIEW

Modules contain Python source code and end with the `(.py)` file suffix. Simple applications may contain only one module, while complex applications may contain many modules. Use lower-case characters when forming module names. Use underscores, if required, to clarify the module name. Name modules according to their responsibility within the application. Add a module named `main.py` to serve as an explicit application entry point.



2 FUNCTIONS — THE 10,000 FOOT VIEW

OK, you may be wondering what goes into a module. Functions, among other things. As you learned in the previous section, modules serve as the primary means for organizing an application's architecture. Functions organize code *within* a module.

You're already familiar with functions. So far in this book, you've encountered a handful of Python's *built-in functions* including `float()`, `input()`, `int()`, `print()`, `repr()`, and `str()`. You'll get the opportunity to use more Python built-in functions as you progress through the book.

In this section, you'll get a high-level overview of the purpose and use of functions and several design considerations to take into account when creating your own functions. This will prepare you for a deeper dive into how to define and employ functions in your programs.

2.1 WHAT IS A FUNCTION?

Good question. The answer depends on your point of view.

2.1.1 CLIENT POINT OF VIEW

If you use a function in your code, all you care about is the function's *name*, *how to call it*, *what data you can pass to it*, and *what data it returns*. In other words, you care only about the function's *interface* and what it does. You should not care *how* the function performs its intended task, nor should you ever *need* to care about how the function performs its intended task.

A function viewed in this manner can be thought of as a black box. The black box represents the function's intended purpose or task. A function's arguments represent the inputs to the black box and its return values represent the outputs. Figure 12-4 illustrates these concepts.

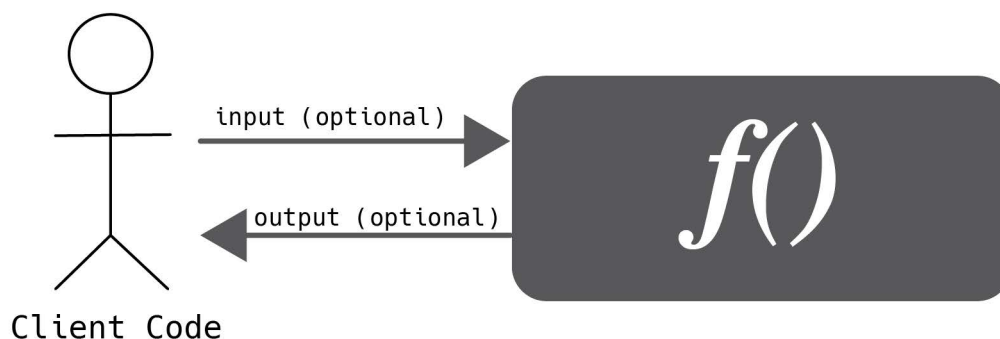


Figure 12-4: Function Viewed as Black Box

2.1.2 DEVELOPER POINT OF VIEW

If you're designing and writing functions, then you need to know how to do so such that users, including yourself, needn't care about how the function performs its intended task.

To you, the function developer, a function is a logical grouping of code accessible by name. The function's name should indicate its intended purpose or task, and all the code contained within the function should support the function's intended task and pull no surprises.

If a function requires input data, then you need to define one or more function *parameters* to convey this data into the function. If the function returns a result, then you must define and implement a data structure suitable to convey the data back to the user.

Functions impose *structure*, *order*, and *clarity* upon the code located within its containing module. Functions provide a means of code *sharing* and *reuse*. Functions make it easier to *think* and *reason* about program behavior. Properly implemented functions make code easier to maintain.

2.2 THE MANY PURPOSES OF A FUNCTION

Functions serve multiple purposes. Primarily, they provide a means to *group*, *name*, and *call* logically-related code segments. By grouping logically-related code into callable segments, you reduce program size and increase code reuse and modularity. Over time, code quality improves as you work out the function’s bugs. (To learn more about the origin of the term “bug”, read: [September 9, 1947 CE: World’s First Computer Bug](#))

Functions increase code clarity, assuming a function’s name indicates its intended purpose or task. Adopting sound function naming practices makes it significantly easier to figure out what a program is supposed to do. Novice developers often can be short sighted when naming functions. I’ve seen students make up function names that offer not the slightest indication of a function’s purpose. Then they wonder why they can’t understand their own code, and as a result, can’t figure out how to fix it!

The moment you realize you’re not naming functions for your own self-gratification, but for the sake of other developers who may come along later and try to decipher how your program works, is the moment you take your first steps towards becoming a professional software engineer. Well-named functions lead to higher-code quality through clarified intentions, and make it easier to spot and correct errors in code.

Functions form an atomic, testable unit. Later in the book, when you learn about unit testing, you’ll learn the importance of writing testable functions. A well-written function is one that can be thoroughly tested with automated test suites. You’ll learn more about automated testing and unit tests later in Chapter 21: Unit Testing. A function that defies automated testing should be redesigned and rewritten.

2.3 FUNCTION DESIGN CONSIDERATIONS

You need to keep in mind several considerations when designing and writing functions. You want your functions to do one thing and do it well. You also want a function to stay in its swim lane. By this I mean the results of a function call must be predictable or *deterministic*; a function call should not mysteriously change some other part of an application surreptitiously. To this end, you want to write functions that *maximize cohesion*, *minimize coupling*, and *minimize side effects*.

2.3.1 MAXIMIZE COHESION

The term *cohesion* means the degree to which a function sticks to its intended purpose. All the code contained within a function should exist for no other purpose than to fulfill the function’s intended purpose. A function’s intended purpose is conveyed primarily by its name. A maximally cohesive function wastes no time or resources executing code not related to its intended purpose. In other words, a highly-cohesive function performs its intended task and nothing but its task.

2.3.2 MINIMIZE COUPLING

The term *coupling* means the degree to which a function is interconnected with other modules or functions. Coupling is a measure of interdependence. Functions can be highly/tightly coupled or loosely coupled. If function `a()` depends on functions `b()`, `c()`, and `d()`, then a programmer assigned to maintain function `a()` must understand functions `b()`, `c()`, and `d()` before they can fully understand function `a()`. Also, as you will learn later when you encounter software design principles, any modifications made to functions `b()`, `c()`, and `d()`, will impact the behavior of function `a()`. In this example, function `a()` is said to depend upon functions `b()`, `c()`, and `d()`.

2.3.3 MINIMIZE SIDE EFFECTS

A side effect is anything a function does not indicated by its name or intended purpose. An example of a side effect might be where a call to a `change_first_name()` function also changes the last name. Another example of a side effect would be calling the same `change_first_name()` function and it also prints a message to the console.

At their worst, undocumented side effects wreak havoc on application state. Some types of side effects are less insidious than others. Printing a message to the console, while not likely to crash an application, is still considered a side effect.

QUICK REVIEW

Functions organize code within a module. A function's name must clearly indicate its intended purpose. A well-designed function can be treated like a black box. A function can receive input data via parameters and return output data in whatever form is required. A function is a first class object in Python. A function can be assigned to a variable, passed into a function as an argument, and returned from a function.

When writing functions, ensure they are maximally cohesive and minimally coupled. A function must preform its intended task and nothing else. A function must be deterministic; it must return the same output for a given set of inputs.

3 DEFINING AND CALLING FUNCTIONS

It's time now to dive deeper into functions. In this section, you'll learn how to define and call functions, how to name functions, how to define function parameters, how to pass data into a function using function arguments, and how to define positional and default parameters. You'll also learn how to return single and multi-valued data from functions.

3.1 DEFINING AND CALLING FUNCTIONS

Before you can use a function in a program, it must be defined within a module. The function can then be used within the same module or the module can be imported into another module that requires its services.

3.1.1 ANATOMY OF A FUNCTION

I'd like to start by showing you the anatomy of a typical function as shown in example 12.1.

12.1 *greetings.py*

```

1  def print_hello_world():
2      print('hello, world')
3

```

Referring to example 12.1 — The definition for the `print_hello_world()` function starts on line 1 of the `greetings.py` module. The keyword `def` begins in the left-most column, followed by a space, then followed by the function name `print_hello_world`. The open and closing parentheses “`()`” signify this is a function definition. Finally, a colon ‘`:`’ is added to indicate the beginning of the function body, which begins on the next line and is indented 1 tab or 4 spaces. Note that PEP-8 recommends spaces.

Generally speaking, any number of lines of code can appear within the function body, but you want to keep function definitions as short as possible for clarity. Long-winded functions can prove difficult to understand and can usually be refactored into several shorter, more cohesive methods.

Note that the `print_hello_world()` function takes no arguments (because it defines no parameters), and returns no data. To see this function run in a program, you need to import the `greetings.py` module into another module that serves as an application entry point as shown in example 12.2.

12.2 *main.py*

```

1  import greetings
2
3  def main():
4      greetings.print_hello_world()
5
6  if __name__ == '__main__':
7      main()
8

```

Referring to example 12.2 — The `main.py` module serves as the application entry point and can be run directly by the Python interpreter. To run the program, use the following command:

```
python3 main.py
```

Figure 12-5 shows the results of running this program.

```

-bash
~/dev/cst_with_python_1st_ed/chapter12/function_anatomy (main)
[527:27] swodog@macos-mojave-test-bed $ python3 main.py
hello, world

```

Figure 12-5: Results of Running Example 12-2

3.1.2 NAMING FUNCTIONS

Like modules, function names can contain lower and upper-case letters `a-z`, `A-Z`, numbers `0-9`, and underscores ‘`_`’. PEP-8 advises to favor the use of all lower-case letters, short function names, and use underscores to separate multi-word names for the sake of clarity.

3.1.2.1 USE VERBS IN FUNCTION NAMES

Functions perform actions. Use verbs to clarify as much as possible what the function is supposed to do. For an example of how to perform a noun-verb analysis on a problem refer to Chapter 4, section titled “Noun - Verb Analysis” on page 126.

3.1.2.2 THE MEANING OF A LEADING UNDERSCORE

A function name can begin with an underscore (called a leading underscore) but doing so has special meaning. It signals to those using the module that the function is only to be used within the module and is not part of the module’s *public interface*. Note that this is merely a Mouseketeer’s agreement since Python does not support data encapsulation and has no notion of private or protected module members found in languages like Java or C#. So, if you see functions that begin with leading underscores, you should not use those functions. Later, you will see how different approaches to importing modules affects the visibility of functions with leading underscores.

Functions names that do not begin with a leading underscore are considered part of a module’s *public interface*. Again, this is all agreed upon with a Moose Lodge secret handshake.

3.1.3 CALLING FUNCTIONS WITHIN THE SAME MODULE

A function defined within a module can be called within the same module. If you try to call the function when the module is loaded, you must do so after the function definition. If you put the call to the function inside another function, the call can come before or after the function definition. Generally speaking, there’s usually no need to call a function when the module is loaded, so this problem should rarely, if ever, arise, but to illustrate the point, take a look at example 12.3.

12.3 greetings.py (v2)

```

1  # This will throw an exception because the function
2  # is called as the module loads before the function
3  # has been defined.
4  print_hello_world()
5
6  # This will work because this doesn't run until
7  # it's explicitly called.
8  def call_print_hello_world():
9      print_hello_world()
10
11  def print_hello_world():
12      print('hello, world')
13
14  # This works fine. The function definition has
15  # already loaded.
16  print_hello_world()
17

```

Referring to example 12.3 — On line 4 an attempt is made to call the `print_hello_world()` function when the module is loaded into the Python interpreter. This causes an error because the function definition does not appear until line 11. If the call to the `print_hello_world()` function appears within the body of another function as it does on line 9, then the call can come before its definition. If you really must call a function when a module loads, then make the call anytime after the function has been defined. The call to `print_hello_world()` on line 16 works just fine.

Example 12.4 gives the modified `main.py` module.

12.4 main.py (v2)

```

1  import greetings
2
3  def main():
4      greetings.print_hello_world()
5      greetings.call_print_hello_world()
6
7  if __name__ == '__main__':
8      main()
9

```

Referring to example 12.4 — If you attempt to run this program with the `greetings.py` module in its current state, you'll see the following error:

Traceback (most recent call last):

```

File "main.py", line 1, in <module> import greetings
File "greetings.py", line 4, in <module> print_hello_world()
NameError: name 'print_hello_world' is not defined

```

To get the program to run properly comment out line 4 of the `greetings.py` module.

3.1.4 CALLING FUNCTIONS FROM AN EXTERNAL MODULE

As you have seen in this section, to call a function defined in one module from another module, the module that contains the function definition must be imported into the module you need to call it from. There are various ways to import modules as shown in example 12.5.

12.5 main.py (v3)

```

1  #import greetings
2  from greetings import print_hello_world
3  from greetings import call_print_hello_world
4
5  def main():
6      greetings.print_hello_world()
7      greetings.call_print_hello_world()
8
9  if __name__ == '__main__':
10     main()
11

```

Referring to example 12.5 — Line 1, commented out, imported the entire `greetings.py` module. This gave access to all members defined within the `greetings.py` module. Lines 2 and 3 explicitly name the members to import from the `greetings.py` module, namely, the functions `print_hello_world()` and `call_print_hello_world()`. Note that you omit the parentheses when you import a function explicitly.

3.1.4.1 AUTOMATICALLY EXCLUDE INTERNAL FUNCTIONS

You can automatically exclude functions that begin with a leading underscore from being exported from a module by using an asterisks `*` when importing. Take a look at the modified `greetings.py` module given in example 12.6.

12.6 greetings.py (v3)

```

1  def call_print_hello_world():
2      print_hello_world()
3
4  def print_hello_world():

```

```

5     print('hello, world')
6
7     def _internal_function():
8         print('Not part of module public interface.')
9

```

Referring to example 12.6 — On line 7, I’ve added a function named `_internal_function()` that begins with a leading underscore. This signifies that this function should only be used within the `greetings.py` module and is not intended to be exported. (i.e., Not part of the module’s public interface.) Using an asterisks to import a module will automatically exclude all functions that begin with a leading underscore as shown in the modified `main.py` module.

12.7 main.py (v4)

```

1     #import greetings
2     #from greetings import print_hello_world
3     #from greetings import call_print_hello_world
4     from greetings import *
5
6     def main():
7         print_hello_world()
8         call_print_hello_world()
9
10    if __name__ == '__main__':
11        main()
12

```

Referring to example 12.7 — On line 4, I’m using an asterisks ‘*’ to import all public interface functions from the `greetings.py` module. The default behavior of the asterisks is to exclude functions that begin with a leading underscore. That’s nice.

The problem, however, with using the asterisks is that you lose the namespace. See lines 7 and 8. Note that the module name `greetings` is no longer required to call the `print_hello_world()` and `call_print_hello_world()` functions. This will only pose a problem if you try to import two modules with the same name. It happens from time-to-time, but not often, and there are ways to mitigate the conflicts if and when they occur.

3.1.4.2 EXPLICITLY NAMING EXPORTS WITHIN A MODULE

The asterisks imports every function from a module that does not begin with an underscore. You learned earlier that function names beginning with a leading underscore are meant for internal module use only, while methods that do not begin with a leading underscore are part of a module’s public interface. You can explicitly define which functions belong to a module’s public interface by adding an `__all__` property to the module. Example 12.8 gives the modified `greetings.py` module.

12.8 greetings.py (v4)

```

1     __all__ = ['print_hello_world']
2
3     def call_print_hello_world():
4         print_hello_world()
5
6     def print_hello_world():
7         print('hello, world')
8
9     def _internal_function():
10        print('Not part of module public interface.')
11

```

Referring to example 12.8 — The `__all__` property defined on line 1 lists ‘`print_hello_world`’ as the only exportable function. Now, when you use the asterisks to import everything from the module, only the `print_hello_world()` function is accessible to the `main.py` module as shown in example 12.9.

12.9 *main.py (v5)*

```

1  #import greetings
2  #from greetings import print_hello_world
3  #from greetings import call_print_hello_world
4  from greetings import *
5
6  def main():
7      print_hello_world()
8
9  if __name__ == '__main__':
10     main()
11

```

Referring to example 12.9 — The only function available for use within the `main.py` module is `print_hello_world()`. I’ll leave it to you as a exercise to experiment with various types of import statements and their effects.

3.1.5 PARTING THOUGHTS

You really need to run the previous examples in Visual Studio Code to see the effects of using the various types of imports. Only then can you see which functions are available for use within the importing module.

Later in the book I’ll show you how to import a module and give it an alias. You’ll want to do this when the module name is really long or there are module name conflicts. When these situations arise you’ll know them when you see them.

As far as which import form I recommend, it’s handy to know which namespace (i.e., module name) a function belongs to as you read through the code, so I generally favor importing the whole module like so:

```
import greetings
```

...or importing specific functions from the module like so:

```
from greetings import print_hello_world
```

...this way when I read through my code I can see at a glance to which module a function belongs:

```
greetings.print_hello_world()
```

However, these are simply my personal preferences. Whatever you decide to do, be consistent. Having said that, adopting a convention that increases the understandability of your code and your clarifies your intentions makes it significantly easier to track down and fix problems.

3.2 QUICK REVIEW

Functions organize code within a module. Name functions using verbs to indicate action. Keep functions highly cohesive and loosely coupled.

4 FUNCTION PARAMETERS AND ARGUMENTS

As you learned earlier, functions can receive input data, process the data, and return the results in the required format. To accept input data, a function must define one or more parameters, which convey the data into the function. In this section, I'll show you how to define function parameters and how to pass arguments to functions during a function call.

Important vocabulary to understand in this section are the terms *parameter* and *argument*. A parameter is a variable used to store and convey data into a function. An argument is the actual data that is passed to a function when it is called. An argument is assigned to a function parameter. The function accesses input data via its parameters. Let's take a look at a simple example.

12.10 *greetings.py* (v5)

```

1  __all__ = ['print_hello_world', 'greet_user']
2
3  def print_hello_world():
4      print('hello, world')
5
6  def greet_user(user_name):
7      print(f'hello, {user_name}')
8
9  def _internal_function():
10     print('Not part of module public interface.')
11

```

Referring to example 12.10 — I've added a new method on line 6 named `greet_user()` which defines one parameter called `user_name` and uses it to formulate a string which is passed as an argument to the `print()` function on line 7.

In this example, I'm using a formatted string, also referred to as an 'f' string, which is formed by prepending the character 'f' to an ordinary string. The presence of one or more pairs of curly braces "{"}" denote placeholders for values within the string. The `user_name` parameter is placed within the curly braces. When the `greet_user()` function is called, the argument passed to it is used to set the value of the `user_name` parameter, which is then used to formulate the string being passed to the `print()` function on line 7. Example 12.11 shows this function in action.

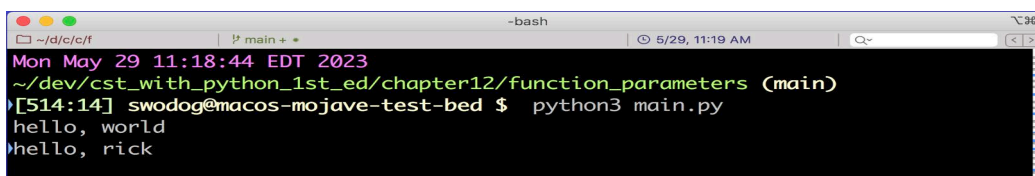
12.11 *main.py* (v6)

```

1  import greetings
2
3  def main():
4      greetings.print_hello_world()
5      greetings.greet_user('rick')
6
7  if __name__ == '__main__':
8      main()
9

```

Referring to example 12.11 — The new `greet_user()` method is called on line 5. The string 'rick' is passed to the function as an argument, which sets the value of the function's `user_name` parameter. Figure 12-6 shows the results of running this program.



```

~ -bash
Mon May 29 11:18:44 EDT 2023
~/dev/cst_with_python_1st_ed/chapter12/function_parameters (main)
[514:14] swodog@macos-mojave-test-bed $ python3 main.py
hello, world
hello, rick

```

Figure 12-6: Results of Running Example 12-11

4.1 FIVE TYPES OF FUNCTION PARAMETERS

Python supports five types of function parameters: *positional-or-keyword*, *positional-only*, *keyword-only*, *var-positional*, and *var-keyword*.

4.1.1 POSITIONAL-OR-KEYWORD

This is the default type of parameter you'll normally use when you define a function in Python. Take a look at the function definition shown in example 12.12.

12.12 *Greetings.py* (v6)

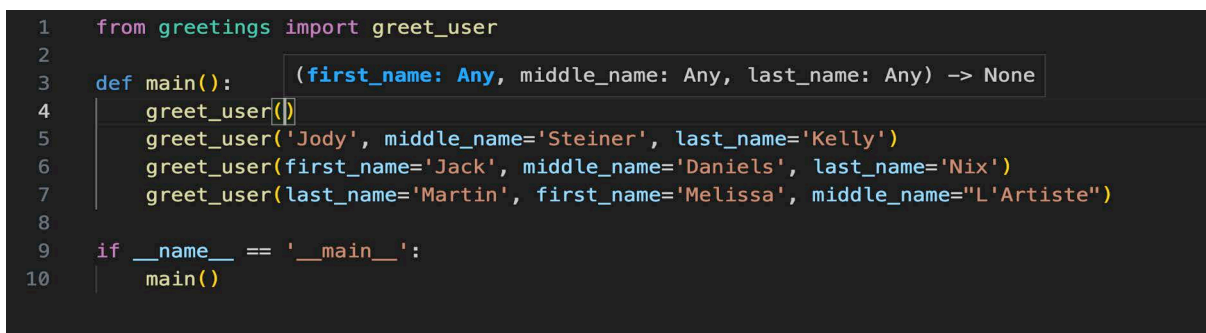
```
1 def greet_user(first_name, middle_name, last_name):
2     print(f'Hello {first_name} {middle_name} {last_name}!')
3
```

Referring to example 12.12 — The `greet_user()` function definition includes three parameters: `first_name`, `middle_name`, and `last_name`, each of which are passed as arguments to the `print()` function on line 2. Example 12.13 shows several ways you can pass arguments to the `greet_user()` function.

12.13 *main.py* (v7)

```
1 from greetings import greet_user
2
3 def main():
4     greet_user('Coralie', 'Sylvia', 'Miller')
5     greet_user('Jody', middle_name='Steiner', last_name='Kelly')
6     greet_user(first_name='Jack', middle_name='Daniels', last_name='Nix')
7     greet_user(last_name='Martin', first_name='Melissa', middle_name="L'Artiste")
8
9 if __name__ == '__main__':
10     main()
11
```

Referring to example 12.13 — Line 4 passes the arguments to positional parameters. The order in which the arguments are passed to the `greet_user()` function dictates the parameter to which they are assigned. Thus, "Coralie" is assigned to `first_name`, "Sylvia" is assigned to `middle_name`, and "Miller" is assigned to the `last_name` parameter. If you were to make a mistake and pass "Miller" first, it would be assigned to the `first_name` parameter. Most IDEs will supply hints about a function's arguments as shown in figure 12-7



```
1 from greetings import greet_user
2
3 def main():
4     greet_user((first_name: Any, middle_name: Any, last_name: Any) -> None)
5     greet_user('Jody', middle_name='Steiner', last_name='Kelly')
6     greet_user(first_name='Jack', middle_name='Daniels', last_name='Nix')
7     greet_user(last_name='Martin', first_name='Melissa', middle_name="L'Artiste")
8
9 if __name__ == '__main__':
10     main()
```

Figure 12-7: VS Code IntelliSense Listing Parameter Names, Position, and Type

Referring to figure 12-7 — Visual Studio Code employs IntelliSense to provide help when entering function arguments. Continuing with example 12.13 — On line 5, the "Jody" string argument is passed to the `first_name` positional parameter, but the "Steiner" and "Kelly" arguments

are assigned to keyword parameters, which take the form *parameter_name=argument*. Note that you can pass all arguments to keyword parameters as shown on lines 6 and 7.

As politicians are fond of saying nowadays, "To be perfectly clear...", function parameters defined as shown in example 12.12 are how you normally define function parameters. Be aware that as soon as you assign an argument to a keyword parameter, the rest of the arguments must be assigned to keyword parameters. This is because the interpreter has no idea as to which parameter a positional argument should be assigned to following the keyword argument. Just something to be aware of. To avoid problematic code, adopt a coding convention and stick with it.

I deal with it this way: When calling functions that define only a few parameters I'll use positional parameters. When calling a function that defines a boat load of parameters, I'll use keyword parameters. That way I'm confident I'm passing the correct argument to the correct parameter.

Pro Tip: Remember — After the initial use of a function keyword parameter, all subsequent arguments must be passed to the function as keyword parameters. When calling functions that define only a few parameters, use positional parameters. When calling functions that require four or more arguments, use keyword parameters to ensure you assign the correct argument to the correct parameter.

4.1.2 POSITIONAL-ONLY

You can force arguments to be passed positional-only by adding a forward slash '/' at some point in the function definition's parameter list as shown in example 12.14.

```
1 def greet_user(first_name, /, middle_name, last_name):
2     print(f'Hello {first_name} {middle_name} {last_name}!')
3
```

12.14 greetings.py (v7)

Referring to example 12.14 — All parameters that appear before the forward slash character are positional-only parameters. Any attempt to assign an argument to `first_name` using keyword parameters will throw an error. Any parameters appearing after the forward slash can be assigned arguments either positionally or via keywords. Example 12.15 shows the valid ways arguments can be passed to the `greet_user()` method.

```
1 from greetings import greet_user
2
3 def main():
4     greet_user('Coralie', 'Sylvia', 'Miller')
5     greet_user('Jody', middle_name='Steiner', last_name='Kelly')
6     greet_user('Jack', 'Daniels', last_name='Nix')
7     #greet_user(last_name='Martin', first_name='Melissa', middle_name="L'Artiste")
8
9 if __name__ == '__main__':
10     main()
11
```

12.15 main.py (v8)

Referring to example 12.15 — Note that on line 4, all arguments are pass positionally. On line 5, the first argument is passed positionally while the last two are passed as keywords. On line 6, the first two arguments are passed positionally while the last argument is passed via keyword. I've commented out line 7 because it will throw and exception since the `first_name` parameter is designated as positional-only yet at attempt is made to use it as a keyword, however, this is not the only reason why this function call will fail. Can you spot another problem with the function call?

4.1.3 KEYWORD-ONLY

Function parameters can be designated as keyword-only by including an asterisks somewhere in the parameter declaration list as shown in example 12.16.

12.16 greetings.py (v8)

```
1 def greet_user(first_name, *, middle_name, last_name):
2     print(f'Hello {first_name} {middle_name} {last_name}!')
3
```

Referring to example 12.16 — The difference between the forward slash '/' and the asterisks '*' is that the forward slash applied to its **preceding** parameters whereas the asterisks applies to its **following** parameters. In the above function definition, an argument can be assigned to the `first_name` parameter either positionally or via keyword, but arguments can only be assigned to the `middle_name` and `last_name` parameters via keywords. Example 12.17 shows the viable ways to pass arguments to the `greet_user()` function.

12.17 main.py (v9)

```
1 from greetings import greet_user
2
3 def main():
4     greet_user('Coralie', middle_name='Sylvia', last_name='Miller')
5     greet_user('Jody', middle_name='Steiner', last_name='Kelly')
6     greet_user(first_name='Jack', middle_name='Daniels', last_name='Nix')
7     greet_user(last_name='Martin', first_name='Melissa', middle_name="L'Artiste")
8
9 if __name__ == '__main__':
10     main()
11
```

Referring to example 12.17 — The `first_name` parameter can be assigned an argument either positionally or via keyword, but the remaining parameters must be assigned argument via keywords.

4.1.4 VAR-POSITIONAL

Sometimes you want to pass a variable number of positional arguments to a function. In times like these you need to use the `*args` parameter as shown in example 12.18.

12.18 greetings.py (v9)

```
1 def greet_user(first_name, *args):
2     print(f'Hello {first_name} ', end='')
3     for s in args:
4         print(f'{s} ', end='')
5     print('!')
6
```

Referring to example 12.18 — This version of the `greet_user()` function allows for people with more than four names. Essentially, the `*args` parameter represents a list of positional values. The first argument passed to the function is assigned to the `first_name` parameter. All subsequent arguments, if any, are passed to the `*args` parameter as a list of arguments. You'll learn more about lists and list processing later in the book. For now, the `for` statement that begins on line 3 iterates over the `args` list and prints each element to the console. Example 12.19 gives the revised `main.py` module.

12.19 main.py (v10)

```
1 from greetings import greet_user
2
3 def main():
4     greet_user('Coralie', 'Sylvia', 'Miller')
```

```

5     greet_user('Jody', 'Steiner', 'Kelly')
6     greet_user('Jack', 'Daniels', 'Nix')
7     greet_user('Melissa', "L'Artiste", 'Martin')
8
9     if __name__ == '__main__':
10        main()
11

```

Referring to example 12.19 — Note that all argument are passed positionally. The first argument is assigned to the `first_name` parameter while the remaining arguments are assigned to the `*args` parameter. Figure 12-8 shows the results of running this program.

```

Sat Aug 19 12:18:28 EDT 2023
~/dev/cst_with_python_1st_ed/chapter12/var_positional_parameters (main)
[532:32] swodog@macos-mojave-test-bed $ python3 main.py
Hello Coralie Sylvia Miller !
Hello Jody Steiner Kelly !
Hello Jack Daniels Nix !
Hello Melissa L'Artiste Martin Miller !

```

Figure 12-8: Results of Running Example 12.19

4.1.5 VAR-KEYWORD

To pass an arbitrary number of keyword arguments to a function use the `**kwargs` parameter. Example 12.20 gives the revised `greet_user()` function.

12.20 greetings.py (v10)

```

1     def greet_user(first_name, **kwargs):
2         print(f'{first_name}', end='')
3         for key, value in kwargs.items():
4             print(f', {key}={value}', end='')
5         print('!')
6

```

Referring to example 12.20 — First, note that the `**kwargs` parameter starts with two asterisks. The `**kwargs` parameter represents a dictionary of key/value pairs. The keys are the parameter keyword names and the values represent the arguments assigned to those keywords. You'll learn more about dictionaries and dictionary processing later in the book. For now just be aware that the `for` statement beginning on line 3 extracts each key/value pair from the dictionary and prints those values to the console. Example 12.21 gives the revised `main.py` module.

12.21 main.py (v11)

```

1     from greetings import greet_user
2
3     def main():
4         greet_user('Coralie', middle_name='Sylvia', last_name='Miller')
5         greet_user('Jody', middle='Steiner', surname='Kelly')
6         greet_user('Jack', middle_name='Daniels', last_name='Nix')
7         greet_user('Melissa', nickname="L'Artiste", maiden_name='Martin', \
8                 last_name='Miller')
9
10    if __name__ == '__main__':
11        main()
12

```

Referring to example 12.21 — Note that the first argument can be passed positionally. The remaining arguments must be passed via keywords, and technically the keywords can be any-

thing, although practically, if you want to do some meaningful processing, you will need to document what keywords to use. Figure 12-9 shows the results of running this program.

```

Sun Oct 6 08:37:41 EDT 2024
~/dev/cst_with_python_1st_ed/chapter12/kw_args_parameters (main)
[507:8] swodog@macos-mojave-testbed $ python3 main.py
Coralie, middle_name=Sylvia, last_name=Miller!
Jody, middle=Steiner, surname=Kelly!
Jack, middle_name=Daniels, last_name=Nix!
Melissa, nickname=L'Artiste, maiden_name=Martin, last_name=Miller!

```

Figure 12-9: Results of Running Example 12-21

4.2 OPTIONAL ARGUMENTS

You can make arguments optional by supplying parameters with default values. Example 12.22 offers a modified `greet_user()` function.

```

1 def greet_user(first_name='John', middle_name='Jay', last_name='Doe'):
2     print(f'hello, {first_name} {middle_name} {last_name}')
3

```

12.22 greetings.py (v11)

Referring to example 12.22 — note that each parameter is assigned a default value. Example 12.23 shows various ways to call the revised method.

```

1 from greetings import greet_user
2
3 def main():
4     greet_user()
5     greet_user('Coralie')
6     greet_user('Steven', 'Bishop')
7     greet_user('Kateryna', '', 'Nesvit')
8
9
10 if __name__ == '__main__':
11     main()
12

```

12.23 main.py (v12)

Referring to example 12.23 — Since all of the parameters defined in the `greet_user()` function have a default value, it can be called with no arguments as shown on line 4. Lines 5 through 7 demonstrate additional ways to call the function. Figure 12-10 shows the results of running this program.

4.3 TYPE HINTS

Python is a dynamically typed language and does not support static, compile-time type checking. The parameters defined by the `greet_user()` function could be assigned any value although such values may not make sense in the context of the function. I'll leave those experiments to you as an exercise. Python does, however, support type hints.

Refer back to figure 12-7 which shows Visual Studio Code's IntelliSense offering help with the `greet_user()` parameters. Notice in that figure how the word "Any" appears next to each

```

Sat Aug 19 17:00:28 EDT 2023
~/dev/cst_with_python_1st_ed/chapter12/optional_arguments (main)
[549:49] swodog@macos-mojave-test-bed $ python3 main.py
hello, John Jay Doe
hello, Coralie Jay Doe
hello, Steven Bishop Doe
hello, Kateryna Nesvit

```

Figure 12-10: Results of Running Example 12.23

parameter hint and "`-> None`" appears at the end of the function. "Any" means the parameters, as defined, can be assigned any type, while "`-> None`" means the function does not return a value.

Example 12.24 shows how to add type hints to a function definition.

```

1 def greet_user(first_name:str, middle_name:str='Jay', last_name:str='Doe') \
2     -> None:
3     print(f'hello, {first_name} {middle_name} {last_name}')
4

```

12.24 greetings.py (v12)

Referring to example 12.24 — Each of the `greet_user()` function parameters has a type hint of `str` meaning arguments passed to the function should be strings. Note, and this is quite important to keep in mind, that type hints are simply that. **They do not enforce typing.** Figure 12-11 shows how IntelliSense looks now.

```

main.py > main
1 from greetings import greet_user
2
3 def main():
4     greet_user('Coralie', 'Jay', 'Doe')
5     greet_user('Kyle', 'Jay', 'Doe')
6     greet_user('Steven', 'Bishop')
7     greet_user('Kateryna', '', 'Nesvit')
8
9
10 if __name__ == '__main__':
11     main()

```

Figure 12-11: IntelliSense Showing String Type Hints along with Return Type of None

4.4 CHECKING AN ARGUMENT'S TYPE

While Python does not perform static typing, you can manually check an argument's type and take action if it's not what you expect. We'll talk more about this later in the book, but in the meantime, take a look at example 12.25.

```

1 def greet_user(first_name:str, middle_name:str='Jay', last_name:str='Doe') \
2     -> None:
3     if isinstance(first_name, str) and isinstance(middle_name, str) and \
4         isinstance(last_name, str):
5         print(f'hello, {first_name} {middle_name} {last_name}')

```

12.25 greetings.py (v13)

```

6     else:
7         raise Exception('Arguments must be strings!')
8

```

Referring to example 12.25 — I'm using the Python built-in function `isinstance()` to check each argument to see if it's a string. If all incoming arguments are strings then line 5 executes, otherwise, line 7 executes and raises an `Exception` with the indicated warning message. Example 12.26 gives the revised `main.py` module.

12.26 `main.py` (v13)

```

1  from greetings import greet_user
2
3  def main():
4      greet_user('Coralie')
5      greet_user('Kyle', )
6      greet_user('Steven', 'Bishop')
7      greet_user('Kateryna', '', 1)
8
9
10 if __name__ == '__main__':
11     main()
12

```

Referring to example 12.26 — Note that on line 7 I'm passing the numeric value 1 in for the last name. Figure 12-12 shows the results of running this program.

```

Sat Aug 19 18:02:11 EDT 2023
~/dev/cst_with_python_1st_ed/chapter12/type_checking (main)
[566:66] swodog@macos-mojave-test-bed $ python3 main.py
hello, Coralie Jay Doe
hello, Kyle Jay Doe
hello, Steven Bishop Doe
Traceback (most recent call last):
  File "/Users/swodog/dev/cst_with_python_1st_ed/chapter12/type_checking/main.py", line 11, in <module>
    main()
  File "/Users/swodog/dev/cst_with_python_1st_ed/chapter12/type_checking/main.py", line 7, in main
    greet_user('Kateryna', '', 1)
  File "/Users/swodog/dev/cst_with_python_1st_ed/chapter12/type_checking/greetings.py", line 9, in greet_user
    raise Exception('Arguments must be strings!')
Exception: Arguments must be strings!

```

Figure 12-12: Results of Running Example 12.26

Referring to figure 12-12 — The first three lines of greetings print to the console, but when the `greet_user()` function is called with the numeric value, it throws an `Exception`. I'll cover exceptions and defensive programming in greater detail throughout the book.

QUICK REVIEW

Python functions can accept input data for processing. Data is passed to a function via arguments. Each argument is assigned to a function parameter for access and processing within the function. Python supports five types of function parameters: *positional-or-keyword*, *positional-only*, *keyword-only*, *var-positional*, and *var-keyword*. Parameters can normally be assigned arguments of any type. To ensure the correct type of data is passed to a function use type hints and verify incoming data via the built-in `isinstance()` method.

5 RETURNING DATA FROM FUNCTIONS

Python functions can return data. Most functions you write will accept arguments, process those arguments, and return the results. Example 12.27 provides a new function called `create_greeting()`.

```

12.27 greetings.py (v14)
1 def create_greeting(first_name:str, middle_name:str='Jay', last_name:str='Doe') \
2     -> str:
3     if isinstance(first_name, str) and isinstance(middle_name, str) and \
4         isinstance(last_name, str):
5         greeting = f'hello, {first_name} {middle_name} {last_name}'
6         return greeting
7     else:
8         raise Exception('All arguments must be valid strings!')
9

```

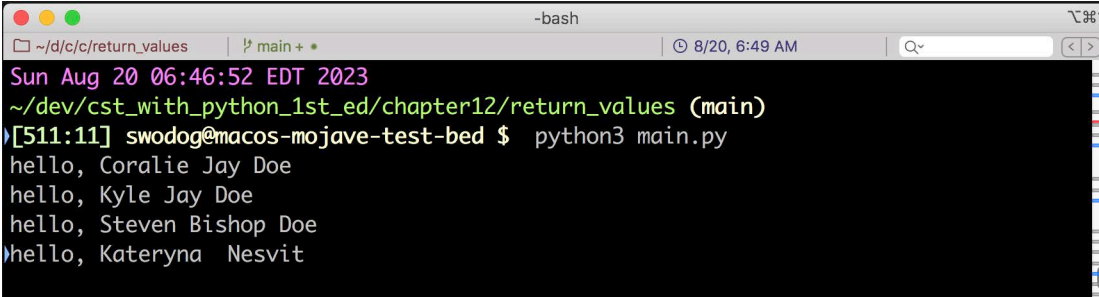
Referring to example 12.27 — The `create_greeting()` function uses type hints to indicate string type objects are to be used as incoming arguments. It also indicates the function returns a string. Next, each parameter's type is verified to be a string with the help of the built-in `isinstance()` function. If all the parameters are strings, they are used to formulate the `greeting` string variable on line 5 which is returned via the `return` statement on line 6. If one of the incoming arguments is anything other than a string, the function raises an exception on line 8 with a helpful message. Example 12.28 shows the `create_greeting()` function in action.

```

12.28 main.py (v14)
1 from greetings import create_greeting
2
3 def main():
4     print(create_greeting('Coralie'))
5     print(create_greeting('Kyle', ))
6     print(create_greeting('Steven', 'Bishop'))
7     print(create_greeting('Kateryna', '', 'Nesvit'))
8
9
10 if __name__ == '__main__':
11     main()
12

```

Referring to example 12.28 — Since the `create_greeting()` method returns a string when it is called, it's return value can be passed into the `print()` function as shown on lines 4 through 7. This is an example of nested function calls. Anywhere an object of a particular type can be used in a program, you can substitute a function call that returns the same type. Figure 12-13 shows the results of running this program.



```

-bash
~/dev/cst_with_python_1st_ed/chapter12/return_values (main)
[511:11] swodog@macos-mojave-test-bed $ python3 main.py
hello, Coralie Jay Doe
hello, Kyle Jay Doe
hello, Steven Bishop Doe
hello, Kateryna Nesvit

```

Figure 12-13: Results of Running Example 12.28

QUICK REVIEW

Python functions can return data. A function can take input data via arguments, process the data, and return the results.

6 FUNCTIONS ARE FIRST-CLASS OBJECTS

In this section you will discover the real power of Python functions and take your first steps towards understanding a concept known as functional programming. Python treats functions as first class objects.

6.1 ASSIGNING TO VARIABLES AND PASSING AS ARGUMENTS

Functions can be assigned to variables and passed as arguments to other functions. Example 12.29 gives the code for a module named `calculator.py`.

12.29 calculator.py

```

1  __all__ = ['calc']
2
3  add = lambda a, b : a + b
4  sub = lambda a, b : a - b
5  mul = lambda a, b : a * b
6  div = lambda a, b : a / b
7
8  def calc(a:float, b:float, op:str) -> float:
9      result = 0
10     match op:
11         case '+': result = _calc(a, b, add)
12         case '-': result = _calc(a, b, sub)
13         case '*': result = _calc(a, b, mul)
14         case '/': result = _calc(a, b, div)
15     return result
16
17 def _calc(a:float, b:float, op:callable) -> float:
18     return op(a,b)
19
```

Referring to example 12.29 — Starting at the top, I've designated the `calc()` method as the only member of the `calculator` module to be exported. The remaining members are private to the module. On lines 3 through 6, I define four variables and assign lambda functions to each one with the indicated operation. For example, I have assigned to the `add` variable on line 3 a function that takes two arguments, `a` and `b` and returns their sum. Skipping down to line 17, I define a function named `_calc()`, which takes two float parameters, `a` and `b`, and a callable parameter named `op`. Finally, on line 8, I define the `calc()` function, which takes two float parameters, `a` and `b`, and a string parameter to indicate which operation is to be performed. In the body of the `calc()` method, I have a `match` statement, which checks the `op` string and executes the indicated operation by calling the `_calc()` method and passing in the corresponding operation variable `add`, `sub`, `mul`, or `div`. Example 12.30 shows the `calc()` method in action.

12.30 main.py

```

1  from calculator import *
2
3  def main():
4      print(f'1 + 2 = {calc(1, 2, "+")}')

```

```

5     print(f'2 - 4 = {calc(2, 4, "-")}')
6     print(f'5 * 5 = {calc(5, 5, "*")}')
7     print(f'14 / 7 = {calc(14, 7, "/")}')
8
9     if __name__ == '__main__':
10        main()
11

```

Referring to example 12.30 — Starting on line 1, using the asterisks to import, I'm only importing the `calc()` method as indicated by the `__all__` directive in the calculator module. On lines 4 through 7, I perform each of the four supported calculator operations: '+', '-', '*', and '/' and print the results to the console. Figure 12-14 shows the results of running this program.

```

Sun Aug 20 08:27:29 EDT 2023
~/dev/cst_with_python_1st_ed/chapter12/functions_first_class_citizens (main)
[528:28] swodog@macos-mojave-test-bed $ python3 main.py
1 + 2 = 3
2 - 4 = -2
5 * 5 = 25
14 / 7 = 2.0

```

Figure 12-14: Results of Running Example 12.30

6.2 PARTING THOUGHTS

In this short section, I've given you just a glimpse of the power of functions. Keep in mind that anything you can do with a function, you can do with a method. A method is a function that belongs to a class. You'll learn more about classes and methods in Part IV: Object-Oriented Programming which includes chapters 17 through 20.

QUICK REVIEW

Functions are first class objects in Python. They can be assigned to variables and passed as arguments to functions.

SUMMARY

Modules contain Python source code and end with the `(.py)` file suffix. Simple applications may contain only one module, while complex applications may contain many modules. Use lower-case characters when forming module names. Use underscores, if required, to clarify the module name. Name modules according to their responsibility within the application. Add a module named `main.py` to serve as an explicit application entry point

Functions organize code within a module. A function's name must clearly indicate its intended purpose. A well-designed function can be treated like a black box. A function can receive input data via parameters and return output data in whatever form is required. A function is a first class object in Python. A function can be assigned to a variable, passed into a function as an argument, and returned from a function.

When writing functions, ensure they are maximally cohesive and minimally coupled. A function must perform its intended task and nothing else. A function must be deterministic; it must return the same output for a given set of inputs.

Functions organize code within a module. Name functions using verbs to indicate action. Keep functions highly cohesive and loosely coupled.

Functions can accept input data for processing. Data is passed to a function via arguments. Each argument is assigned to a function parameter for access and processing within the function. Python supports five types of function parameters: *positional-or-keyword*, *positional-only*, *keyword-only*, *var-positional*, and *var-keyword*. Parameters can normally be assigned arguments of any type. To ensure the correct type of data is passed to a function use type hints and verify incoming data via the built-in `isinstance()` method.

Functions can return data. A function can take input data via arguments, process the data, and return the results.

Functions are first class objects in Python. They can be assigned to variable and passed as arguments to functions.

SKILL-BUILDING EXERCISES

1. **Run Chapter Examples:** Run the example code listed in this chapter. Experiment. Change the code, run it, and note the effects.
2. **Historical Research:** Read chapters 1 - 9 of the classic computer science text *Structured Design by Edward Yourdon and Larry Constantine*.
3. **Dive Deeper Into Functions:** Consult the Python documentation at <https://docs.python.org/3/> and focus on how to name, declare, and call functions.
4. **Dive Deeper Into Lambda Functions:** Example 12.19 featured lambda functions being assigned to variables. Perform the following Google search and explore the topic of Python lambda functions: <https://www.google.com/search?client=safari&rls=en&q=%22python+lambda%22+file:pdf+site:org&ie=UTF-8&oe=UTF-8>. Answer the following question: What's the difference between ordinary functions defined with the `def` keyword and lambda functions?
5. **Coding Exercise:** Modify example 12.19 and change the lambda functions to ordinary functions defined with the `def` keyword. What changes did you have to make to get things to run properly?
6. **Coding Exercise:** Modify example 12.19 and enable a variable number of numeric operands to be passed to the function using the `*args` parameter.
7. **Research Coupling and Cohesion:** Study the terms coupling and cohesion as they apply to modules and functions. Explain why it is important to maximize cohesion and minimize coupling. What problems are software engineers attempting to solve by adhering to these basic

principles?

8. **Dive Deeper Into Functions:** Explain the difference between a parameter and an argument.
9. **Coding Exercise:** Write a program that takes a variable number of numeric arguments and returns their sum.
10. **Coding Exercise:** Write a program that takes a variable number of numeric arguments and returns their mean.

SUGGESTED PROJECTS

Perform the following analysis and design activities for each of the suggested projects in this section:

- Create a list of primary application requirements
- Create a list of modules required to fulfill the requirements
- Assign requirements and responsibilities to each module
- Draw the dependency relationship between application modules
- Create a list of functions within each module that will fulfill the module's requirements and responsibilities
- Assign specific responsibilities to each function
- For each function identify required input data and output data
- Name modules and functions in accordance with the *PEP-8 Style Guide*

1. **Package Tracking Application:** Large events set up package receiving and distribution areas at venues to receive incoming packages for vendors and attendees. Packages are stored on the floor in a grid system. Typical grid coordinates include A - Z and 1 - 26, so, for example, an incoming package might be placed on grid A1. Multiple packages to the same addressee can occupy a grid location. Persist data to either a file in JSON format or to a database.
2. **Item Inventory Application:** In the event of catastrophe, insurance agencies require proof of ownership before paying for lost or destroyed items. Create an inventory application that records the item name, description, cost, quantity, and location. It's even better if you can attach a picture of the item and a receipt showing how much you paid. Persist data to either a file in JSON format or to a database.
3. **Document Organization Application:** Design an application that scans your computer for files with the following suffixes: doc, docx, xls, xlsx, jpg, png, pdf, and any other you may be interested in tracking. Save the absolute path to the file in a data store along with a description of the file, date created, and any other metadata about the file you desire. Persist data to either a file in JSON format or to a database.
4. **Web News Scraper And Archiver:** You are interested in the news but don't have time to manually scan the Internet for articles of interest. Design an application that scrapes the web for articles based on topic. You'll need to do some research regarding how data is presented on

each website you intend to scrape. Persist data to either a file in JSON format or to a database.

5. **Notes Application:** Design an application that lets you take and save notes. Organize notes by topic, class, and date. Persist data to either a file in JSON format or to a database.
6. **Food Ordering Application:** Design an application that a restaurant might use to log incoming food orders. Record menu items ordered, customer details including address, phone number, and email. Enable delivery scheduling. Provide a reporting feature that lets users see sales by item, store, or time period. (i.e., hourly, daily, monthly, quarterly, or custom date range)
7. **Amazon Price Tracking Application:** Design an application that tracks price information for one or more items for sale on Amazon. Add price change alerts to be sent to registered email addresses or via text messages.
8. **Recipe and Shopping List Application:** Design an application that stores and displays recipes and required ingredients. Enable the application to generate a shopping list from selected recipes.
9. **Heavy Traffic Alert Application:** Design an application that monitors traffic along designated routes or within designated areas. Enable alerting when traffic density increases, jams, or declines. Research publicly available APIs from which you can obtain traffic data.
10. **Data Analysis Application:** Visit <https://data.gov>, select a data set, and design an application to download, analyze, and visually your analytic results.

SELF-TEST QUESTIONS

1. What's the purpose of a module?
2. What happens to the name of a module when you run it directly with the Python interpreter?
3. Which PEP would you consult for questions regarding Python naming conventions?
4. What's the purpose of a function?
5. What is meant by the term Black Box with regards to functions?
6. What character would you use to indicate a function is meant not to be exported or used by client code?
7. What's the purpose of the `__all__` property?
8. What character would you use to designate positional-only function parameters?

9. What parameter would you use to pass in a variable amount of positional arguments?
10. What parameter would you use to pass in a variable amount of keyword arguments?

REFERENCES

Structured Design by Edward Yourdon and Larry Constantine, https://vtda.org/books/Computing/Programming/StructuredDesign_EdwardYourdonLarryConstantine.pdf

Python Documentation, Latest Version: <https://docs.python.org/3/>

Government Publication NBSIR 75-780, Mathematics and Engineering in Computer Science, <https://www.govinfo.gov/content/pkg/GOVPUB-C13-3fa03b5da58fa1f9872190856cdccef3/pdf/GOVPUB-C13-3fa03b5da58fa1f9872190856cdccef3.pdf>

The Impact of Component Modularity on Design Evolution: Evidence from the Software Industry, https://www.hbs.edu/ris/Publication%20Files/08-038_187f1243-7d1e-464a-bddd-bf7c09873284.pdf

Size and Cohesion Metrics as Indicators of the Long Method Code Smell: An Empirical Study, Charalampidou, et. al., ACM Digital Library, <https://dl.acm.org/doi/10.1145/2810146.2810155>

Misconceptions of the Agile Zealots, by Edward V. Berard, The Object Agency, L.L.C., <https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=1684bc535f5b81ad0bc2ea11ce94471027df1089>

Mapping the Cognitive Demands of Learning to Program, 1409-BK — pp. 555-563 — Contributors to Thinking — brw — 7/30/86, https://web.stanford.edu/~roypea/RoyPDF%20folder/A44_Kurland_etal_*REDO.pdf

Wegner, "Programming Languages—The First 25 Years," in *IEEE Transactions on Computers*, vol. C-25, no. 12, pp. 1207-1225, Dec. 1976, doi: 10.1109/TC.1976.1674589.

PEP 8 - Style Guide for Python Code, <https://peps.python.org/pep-0008/>

NOTES
