Lesson 21. Achieving modularity and abstraction with functions

After reading lesson 21, you'll be able to

- Write code that uses functions
- Write functions with (zero or more) parameters
- Write functions that (may or may not) return a specified value
- Understand how variable values change in different function environments

In lesson 20, you saw that dividing larger tasks into modules can help you to think about problems. The process of breaking up the task leads to two important ideas: modularity and abstraction. *Modularity* is having smaller (more or less independent) problems to tackle, one by one. *Abstraction* is being able to think about the modules themselves at a higher level, without worrying about the details of implementing each. You already do this a lot in your day-to-day life; for example, you can use a car without knowing how to build one.

These modules are most useful for decluttering your larger tasks. They *abstract* certain tasks. You have to figure out the details of how to implement a task only once. Then you can reuse a task with many inputs to get outputs without having to rewrite it again.

Consider this

For each of the following scenarios, figure out what set of steps would make sense to abstract into a module:

- You're in school. Your teacher is taking attendance. She calls a name. If the student is there, that student says their name. Repeat this process for every student taking that class.
- A car manufacturer is assembling 100 cars a day. The assembly process is made up of (1) assembling the frame, (2) putting in the engine, (3) adding the electronics, and (4) painting the body. There are 25 red, 25 black, 25 white, and 25 blue cars.

Answer:

- Module: Call a name
- Module: Assemble the frame Module: Put in the engine Module: Add electronics Module: Paint each car

21.1. WRITING A FUNCTION

In many programing languages, a *function* is used to stand for a module of code that achieves a simple task. When you're writing a function, you have to think about three things:

- What input the function takes in
- What operations/calculations the function does
- What the function returns

Recall the attendance example from the preceding "Consider this" exercise. Here's a slight modification of that situation, along with one possible implementation using functions. Functions start with a keyword `def`. Inside the function, you document what the function does between triple quotes—mention what the inputs are, what the function does, and what the function returns. Inside the function shown in listing 21.1, you check whether every student in the classroom roster is also physically present in the classroom, with a `for` loop. Anyone matching this criteria gets their name printed. At the end of the `for` loop, you return the string `finished taking attendance`. The word `return` is also a keyword associated with functions.

```
def take_attendance(classroom, who_is_here):
    """
    classroom, tuple
    who_is_here, tuple
    Checks if every item in classroom is in who_is_he
    And prints their name if so.
    Returns "finished taking attendance"
    """
    for kid in classroom:
        if kid in who_is_here:
            print(kid)
    return "finished taking attendance"
```

- ***1*** **Function definition**
- ***2*** **Function specification (docstring)**
- ***3*** **Loops over every student in class**
- ***4*** **Checks whether the student is also in the who_is_here tuple**
- ***5*** **Prints the name of the kid who's here**
- ***6*** **Returns a string**

Listing 21.1 shows a Python function. When you tell Python that you want to define a function, you use the `def` keyword. After the `def` keyword, you name your function. In this example, the name is `take_attendance`. Function names abide by the same rules that variables do. After the function name, you put in parentheses all inputs to the function, separated by a comma. You end the function definition line with a colon character.

How does Python know which lines of code are part of a function and which aren't? All lines that you want to be a part of the function are indented—the same idea used with loops and conditionals.

Quick check 21.1

Write a line to define functions with the following specifications:

1

A function named `set_color` that takes in two inputs: a string named `name` (representing the name of an object) and a string named `color` (representing the name of a color)

2

A function named `get_inverse` that takes in one input: a number named `num`

3

A function named `print_my_name` that doesn't take in any inputs

21.1.1. Function basics: what the function takes in

You use functions to make your life easier. You write a function so you can reuse its guts with different inputs. This allows you to avoid having to copy and paste the implementation with only a couple of variable values changed.

All inputs to a function are variables called *parameters*, or *arguments*. More specifically, they're called *formal parameters*, or *formal arguments*, because inside the function definition these variables don't have any value. A value is assigned to them only when you make a call to a function with some values, which you'll see how to do in a later section.

Quick check 21.2

For the following function definitions, how many parameters does each take in?

1

```
def func_1(one, two, three):
```

2

```
def func_2():
```

3

21.1.2. Function basics: what the function does

When you write the function, you write the code inside the function assuming that you have values for all the parameters to the function. The implementation of a function is just Python code, except it starts out indented. Programmers can implement the function in any way they want.

Quick check 21.3

Are the bodies of each of the following functions written without any errors?

1

```
def func_1(one, two, three):
    if one == two + three:
        print("equal")
```

2

```
def func_2():
    return(True and True)
```

3

```
def func_3(head, shoulders, knees):
return "and toes"
```

21.1.3. Function basics: what the function returns

Functions should do something. You use them to repeat the same action on a slightly different input. As such, function names are generally descriptive action words and phrases: `get_something`, `set_something`, `do_something`, and others like these.

Quick check 21.4

Come up with an appropriate name for functions that do each of the following:

1

A function that tells you the age of a tree

2

A function that translates what your dog is saying

3

A function that takes a picture of a cloud and tells you the closest animal it resembles

4

A function that shows you what you'll look like in 50 years

A function creates its own environment, so all variables created inside this environment aren't accessible anywhere outside the function. The purpose of a function is to perform a task and pass along its result. In Python, passing results is done using the `return` keyword. A line of code that contains the `return` keyword indicates to Python that it has finished with the code inside the function and is ready to pass the value to another piece of code in the larger program.

In listing 21.2, the program concatenates two string inputs together and returns the length of the resulting concatenation. The function takes in two strings as parameters. It adds them and stores the concatenation into the variable named `word`. The function returns the value `len(word)`, which is an integer corresponding to the length of whatever value the variable `word` holds. You're allowed to write code inside the function after the `return` statement, but it won't be executed.

Listing 21.2. A function to tell you the length of the sum

```
def get_word_length(word1, word2):          1
    word = word1+word2                      2
    return len(word)                        3
    print("this never gets printed")        4
```

- **1 Function definition; takes in two parameters**
- **2 Concatenates two parameters**
- **3 Return statement; returns the length of the concatenation**
- **4 Nothing after the return statement is executed**

Quick check 21.5

What does each function return? What is the type of the return variable?

1

```
def func_1(sign):
    return len(sign)
```

2

```
def func_2():
    return (True and True)
```

3

```
def func_3(head, shoulders, knees):
    return("and toes")
```

21.2. USING FUNCTIONS

In section 21.1, you learned how to define a function. Defining a function in your code only tells Python that there's now a function with this name that's going to do something. The function doesn't run to produce a result until it's called somewhere else in the code.

Assume that you've defined the function `word_length` in your code, as in listing 21.2. Now you want to use the function to tell you the number of letters in a full name. Listing 21.3 shows how to call the function. You type in the name of the function and give it *actual parameters*—variables that have a value in your program. This is in contrast to the formal parameters you saw earlier, which are used when defining the function.

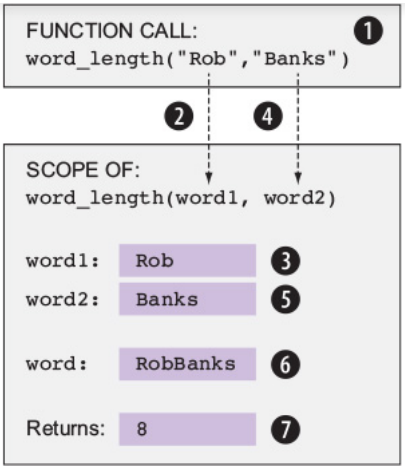
Listing 21.3. How to make a call to a function

```
def word_length(word1, word2):
    word = word1+word2
    return len(word)
    print("this never gets printed")

length1 = word_length("Rob", "Banks")
length2 = word_length("Barbie", "Kenn")
length3 = word_length("Holly", "Jolley")

print("One name is", length1, "letters long.")
print("Another name is", length2, "letters long.")
print("The final name is", length3, "letters long.")
```

- **1 Function definition code for word_length**
- **2 Each line calls the function with different inputs and assigns the return from the function to a variable.**
- **3 Each line prints the variables.**

Figure 21.1 shows what happens with the function call `word_length("Rob", "Banks")`. A new *scope* (or environment) is created whenever a function call is made and is associated with that specific function call. You can think of the scope as a separate mini-program that contains its own variables, not accessible to any other part of the program.

Figure 21.1. What happens when you make a function call in 1? With 2 and 3, the first parameter is mapped in the function's scope. With 4 and 5, the second parameter is mapped. 6 is another variable created inside the function. 7 is the return value. After the function returns, the function scope disappears along



After the scope is created, every actual parameter is mapped to the function's formal parameter, preserving the order. At this point, the formal parameters have values. As the function progresses and executes its statements, any variables that are created exist only in the scope of this function call.

21.2.1. Returning more than one value

You may have noticed that a function can return only one object. But you can "trick" the function into returning more than one value by using tuples. Each item in the tuple is a different value. This way, the function returns only one object (a tuple), but the tuple has as many different values (through its elements) as you want. For example, you can have a function that takes in the name of a country and returns one tuple whose first element is the latitude of the country's center and whose second element is the longitude of the country's center.

Then, when you call the function, you can assign each item in the returned tuple to a different variable, as in the following listing. The function `add_sub` adds and subtracts the two parameters, and returns one tuple consisting of these two values. When you call the function, you assign the return result to another tuple `(a,b)` so that `a` gets the value of the addition and `b` gets the value of the subtraction.

Listing 21.4. Returning a tuple

```
def add_sub(n1, n2):
    add = n1 + n2
    sub = n1 - n2
    return (add, sub)          1

(a, b) = add_sub(3,4)          2
```

- ***1* Returns a tuple with addition and subtraction values**
- ***2* Assigns result to a tuple**

Quick check 21.6

1

Complete the following function that tells you whether the number and suit match the secret values and the amount won:

```
def guessed_card(number, suit, bet):
    money_won = 0
    guessed = False
    if number == 8 and suit == "hearts":
       money_won = 10*bet
       guessed = True
    else:
       money_won = bet/10
    # write one line to return two things:
    # how much money you won and whether you
    # guessed right or not
```

2

Use the function you wrote in (1). If executed in the following order, what do the following lines print?

- `print(guessed_card(8, "hearts", 10))`
- `print(guessed_card("8", "hearts", 10))`
- `guessed_card(10, "spades", 5)`

```
(amount, did_win) = guessed_card("eight", "hea
print(did_win)
print(amount)
```

21.2.2. Functions without a return statement

You may want to write functions that print a message and don't explicitly return any value. Python allows you to skip the `return` statement inside a function. If you don't write a `return` statement, Python automatically returns the value `None` in the function. `None` is a special object of type `NoneType` that stands for the absence of a value.

Look through the example code in listing 21.5. You're playing a game with kids. They're hiding, and you can't see them. You call their names in order:

- If they come out from their hiding spot to you, that's like having a function that returns an object to the caller.
- If they yell out "here" and don't show themselves, that's like having a function that doesn't return the kid object but does print something to the user. You need to get an object back from them, so they all agree that if they don't show themselves, they throw at you a piece of paper with the word *None* written on it.

Listing 21.5 defines two functions. One prints the parameter given (and implicitly returns `None`). Another returns the value of the parameter given. There are four things going on here:

- The first line in the main program that's executed is `say_name("Dora")`. This line prints `Dora` because the function `say_name` has a `print` statement inside it. The result of the function call isn't printed.
- The next line, `show_kid("Ellie")`, doesn't print anything because nothing is printed inside the function `show_kid`, nor is the result of the function call printed.
- The next line, `print(say_name("Frank"))`, prints two things: `Frank` and `None`. It prints `Frank` because the function `say_name` has a `print` statement inside it. `None` is printed because the function `say_name` has no return statement (so by default returns `None`), and the result of the return is printed with the `print` around `say_name("Frank")`.
- Finally, `print(show_kid("Gus"))`, prints `Gus` because `show_kid` returns the name passed in, and the `print` around `show_kid("Gus")` prints the returned value.

Listing 21.5. Function with and without a return

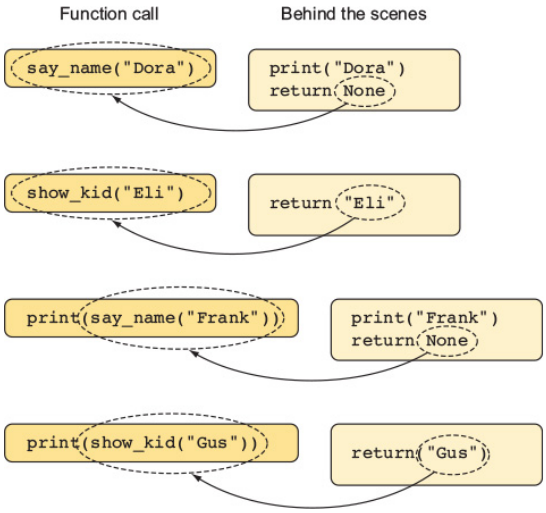
```
def say_name(kid):                    1
    print(kid)                        2

def show_kid(kid):                    3
    return kid                        4

say_name("Dora")                      5
show_kid("Ellie")                     6
print(say_name("Frank"))              7
print(show_kid("Gus"))                8
```

- ***1* Takes in a string with kid name**
- ***2* Doesn't explicitly return anything, so Python returns None**
- ***3* Takes in a string with kid name**
- ***4* Returns a string**
- ***5* Prints "Dora" to the console**
- ***6* Doesn't print anything to the console**
- ***7* Prints Frank, then None, to the console**
- ***8* Prints Gus to the console**

One particularly interesting line is `print(say_name("Frank"))`. The function call itself prints the name of the kid, `Frank`. Because there's no `return` statement, Python returns `None` automatically. The line `print(say_name("Frank"))` is then replaced with the return value to give `print(None)`, which then prints the value `None` to the console. It's important to understand that `None` isn't an object of type `string`. It's the only value for an object of type `NoneType`. Figure 21.2 shows which returned value replaces each function call.

Figure 21.2. Four combinations, representing calling a function that doesn't return a value, calling a function that returns something, printing the result of calling a function that doesn't return a value, and printing the result of calling a function that returns something. The black box is the function call, and the gray box is what

`return None` is automatically added. The black dotted line indicates the value that will replace the function call.



Quick check 21.7

Given the following function and variable initializations, what will each line print, if executed in the following order?

```
def make_sentence(who, what):
    doing = who+" is "+what
    return doing

def show_story(person, action, number, thing):
    what = make_sentence(person, action)
    num_times = str(number) + " " + thing
    my_story = what + " " + num_times
    print(my_story)

who = "Hector"
what = "eating"
thing = "bananas"
number = 8
```

1

```
sentence = make_sentence(who, thing)
```

2

```
print(make_sentence(who, what))
```

3

```
your_story = show_story(who, what, number, thing)
```

4

```
my_story = show_story(sentence, what, number, thing)
```

5

```
print(your_story)
```

21.3. DOCUMENTING YOUR FUNCTIONS

In addition to functions being a way to modularize your code, they're also a way to abstract chunks of code. You saw how abstraction is achieved by passing in parameters so that the function can be used more generally. Abstraction is also achieved through function *specifications*, or *docstrings*. You can quickly read the docstrings to get a sense of what inputs the function takes in, what it is supposed to do, and what it returns. Scanning the text of a docstring is much quicker than reading the function implementation.

Here's an example of a docstring for a function whose implementation you saw in listing 21.1:

```
def take_attendance(classroom, who_is_here):
    """
    classroom: tuple of strings
```

```
    Returns a string, "finished taking attendance"
    """
```

A docstring starts inside the function, indented. The triple quotes """ denote the start and end of the docstring. A docstring includes the following:

- Each input parameter name and type
- A brief overview of what the function does
- The meaning of the return value and the type

SUMMARY

In this lesson, my objective was for you to write simple Python functions. Functions take in input, perform an action, and return a value. They're one way that you can write reusable code in your programs. Functions are modules of code written in a generic way. In your programs, you can call a function with specific values to give you back a value. The returned value can then be used in your code. You write function specifications to document your work so that you don't have to read an entire piece of code to figure out what the function does. Here are the major takeaways:

- Function definitions are just that—definitions. The function is executed only when it's called somewhere else in the code.
- A function call is replaced with the value returned.
- Functions return one object, but you can use tuples to return more than one value.
- Function docstrings document and abstract details of the function implementation.

Let's see if you got this...

Q21.1

1. Write a function named `calculate_total` that takes in two parameters: a float named `price`, and an integer named `percent`. The function calculates and returns a new number representing the price plus the tip: total = price + percent * price.
2. Make a function call to your function with a price of 20 and a percent of 15.
3. Complete the following code in a program to use your function:

```
my_price = 78.55
my_tip = 20
# write a line to calculate and save the n
# write a line to print a message with the
```