Lesson 22. Advanced operations with functions

After reading lesson 22, you'll be able to

- Pass functions (as an object) as a parameter to another function
- Return a function (as an object) from another function
- Understand which variables belong to which scope based on certain rules

Before formally learning about functions in lesson 21, you saw and used functions in simple code. Here are some of the functions you've been using already:

- `len()`—For example, `len("coffee")`
- `range()`—For example, `range(4)`
- `print()`—For example, `print("Witty message")`
- `abs(),sum(),max(),min(),round(),pow()`—For example, `max(3,7,1)`
- `str(),int(),float(),bool()`—For example, `int(4.5)`

Consider this

For each of the following function calls, how many parameters does the function take in, and what's the value of type returned?

- `len("How are you doing today?")`
- `max(len("please"), len("pass"), len("the"), len("salt"))`
- `str(525600)`
- `sum((24, 7, 365))`

Answer:

- Takes in one parameter, returns `24`
- Takes in four parameters, returns `6`
- Takes in one parameter, returns `"525600"`
- Takes in one parameter, returns `396`

22.1. THINKING ABOUT FUNCTIONS WITH TWO HATS

Recall that a function definition defines a series of commands that can be called later in a program with different inputs. Think of this idea like building a car and driving a car: someone has to build a car in the first place, but after they build it, the car sits in a garage until someone wants to use it. And someone who wants to use the car doesn't need to know how to build one, and that person can use it more than once. It may help to think about functions from two perspectives: someone writing a function, and someone who wants to use a function. Sections 22.1.1 and 22.1.2 briefly review the main ideas you should have picked up in the previous lesson.

22.1.1. Writer hat

You write the function in a general way so that it can work with various values. You generalize the function by pretending that the inputs given are named variables. The inputs are called *formal parameters*. You do the operations inside the function, assuming you have values for these parameters.

Parameters and variables defined inside a function exist only in the scope (or environment) of the function. The function scope exists from the time that the function is called to the time that the function returns a value.

The way you abstract a module is by a function specification, or *docstring*. A

Find answers on the fly, or master something new. Subscribe today. See pricing options.

Inside the docstring, you typically write (1) what inputs the function is supposed to take in and their type, (2) what the function is supposed to do, and (3) what the function returns. Assuming the inputs are according to the specification, the function is assumed to behave correctly and guaranteed to return a value according to the specification.

22.1.2. User hat

Using a function is easy. A function is called in another statement in your main program code. When you call a function, you call it with values. These values are the *actual parameters* and replace the function's formal parameters. The function performs the operations it's supposed to by using the actual parameter values.

The output from the function is what the function returns. The function return is given back to whichever statement called the function. The expression for the function call is replaced with the value of the return.

22.2. FUNCTION SCOPE

The phrase "what happens in Vegas stays in Vegas" is an accurate representation of what happens behind the scenes during a function call; what happens in a function code block stays in a function code block. Function parameters exist only within the scope of the function. You can have the same name in different function scopes because they point to different objects. You get an error if you try to access a variable outside the function in which it's defined. Python can be in only one scope at a time and knows only about variables whose scope it's currently in.

22.2.1. Simple scoping example

It's possible for a function to create a variable with the same name as another variable in another function, or even in the main program. Python knows that these are separate objects; they just happen to have the same name.

Suppose you're reading two books, and each has a character named *Peter*. In each book, Peter is a different person, even though you're using the same name. Take a look at listing 22.1. The code prints two numbers. The first number is 5, and the second is 30. In this code, you see two variables named `peter` defined. But these variables exist in different scopes: one in the scope of the function `fairy_tale`, and one in the main program scope.

Listing 22.1. Defining variables with the same name in different scopes

```
def fairy_tale():              1
    peter = 5                  2
    print(peter)               3

peter = 30                     4
fairy_tale()                   5
print(peter)                   6
```

- ***1* Function definition**
- ***2* Variable inside function named peter with a value of 5**
- ***3* Prints 5**
- ***4* First line to execute inside main program, creates a variable named peter with a value of 30**
- ***5* Function call that creates a new scope, prints 5, then the call returns None and the scope ends**
- ***6* Prints 30**

22.2.2. Scoping rules

Here are the rules for deciding which variable to use (if you have more than one with the same name in your program):

- Look in the current scope for a variable with that name. If it's there, use that variable. If it's not, look in the scope of whatever line called the function. It's possible that another function called it.
- If there's a variable with that name in the caller's scope, use that.
- Successively keep looking in outer scopes until you get to the main program scope, also called the *global scope*. You can't look further outside of the global scope. All variables that exist in the global scope are called *global variables*.
- If a variable with that name isn't in the global scope, show an error that the variable doesn't exist.

The code in the next four listings show a few scenarios for variables with the same name in different scopes. There are a couple of interesting things to note:

- You can *access* a variable inside a function without defining it inside a function. As long as a variable with that name exists in the main program scope, you won't get an error.
- You can't *assign* a value to a variable inside a function without defining

Find answers on the fly, or master something new. Subscribe today. See pricing options.

In the following listing, function `e()` shows that you can create and access a new variable with the same name as a variable in your global scope.

Listing 22.2. A function that initializes a variable

```
def e():
    v = 5
    print(v)              1

v = 1
e()                       2
```

- **1 Uses v from function**
- **2 Function call OK; uses v inside the function**

In the next listing, function `f()` shows that it's OK to access a variable even if it's not created inside the function, because a variable of that same name exists in the global scope.

Listing 22.3. A function that accesses a variable outside its scope

```
def f():
    print(v)              1

v = 1
f()                       2
```

- **1 Access variables outside scope**
- **2 Function call OK; uses v from the program**

In the following listing, function `g()` shows that it's OK to do operations with variables not defined in the function, because you're only accessing their values and not trying to change them.

Listing 22.4. A function that accesses more than one variable outside its scope

```
def g():
    print(v+x)            1

v = 1
x = 2
g()                       2
```

- **1 Access only variables**
- **2 Function call OK; uses v and x from the global scope**

In the following listing, function `h()` shows that you're trying to add to the value to a variable inside the function without defining it first. This leads to an error.

Listing 22.5. A function that tries to modify a variable defined outside its scope

```
def h():
    v += 5                1

v = 1
h()                       2
```

- **1 Performs an operation on variable v before defining it inside the function**
- **2 Function call gives an error**

Functions are great because they break your problem into smaller chunks rather than having hundreds or thousands of lines to look at all at once. But functions also introduce scope; with this, you can have variables with the same name in different scopes without them interfering with each other. You need to be mindful of the scope you're currently looking at.

Thinking like a programmer

You should start to get in the habit of *tracing* through a program. To trace through a program, you should go line by line, draw the scope you're in, and write any variables and their values currently in the scope.

The following listing shows a simple function definition and a couple of function calls. In the code, you have one function that returns `"odd"` if a number is odd, and `"even"` if it's even. The code within the function doesn't print anything, it only returns the result. The code starts running at

Find answers on the fly, or master something new. Subscribe today. See pricing options.

ates a scope and maps the value 4 to the formal parameter in the function definition. You do all the calculations and return `"even"` because the remainder when 4 is divided by 2 is 0. `print(odd_or_even(num))` prints the returned value, `"even"`. After printing, you calculate `odd_or_even(5)`. The return from this function call isn't used (isn't printed), and no operations are performed on it.

Listing 22.6. Functions showing different scoping rules

```
def odd_or_even(num):                    1
    num = num%2                          2
    if num == 1:
        return "odd"
    else:
        return "even"

num = 4                                  3
print(odd_or_even(num))                  4
odd_or_even(5)                           5
```

- ***1*** **Function definition takes one parameter, num**
- ***2*** **Remainder when num is divided by 2**
- ***3*** **Variable in global scope**
- ***4*** **A function call that prints its return**
- ***5*** **A function call that doesn't do anything with its return**

Figure 22.1 shows how you might draw a trace of a program. The one tricky thing is that you have two variables named `num`. But because they're in different scopes, they don't interfere with each other.

Quick check 22.1

For the following code, what will each line print?

```
def f(a, b):
    x = a+b
    y = a-b
    print(x*y)
    return x/y

a = 1
b = 2
x = 5
y = 6
```

1

```
print(f(x, y))
```

2

```
print(f(a, b))
```

3

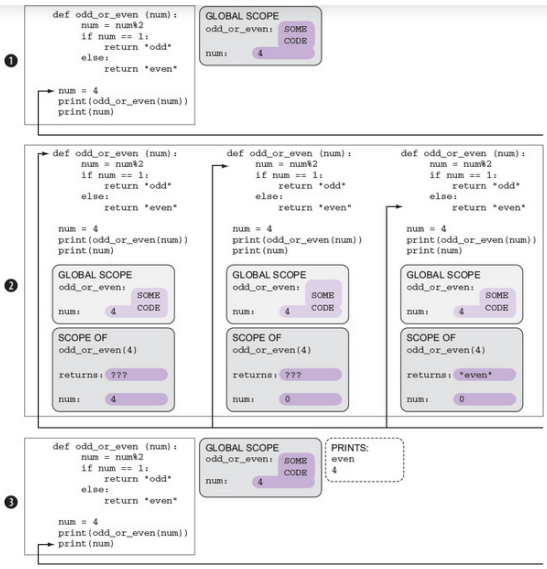
```
print(f(x, a))
```

4

```
print(f(y, b))
```

Figure 22.1. ❶ A trace of a program that indicates whether a number is odd or even. At each line, you draw the scope you're in along with all variables existing in that scope. In ❶ you start the program and are at the line with the arrow. After that line executes, the scope of the program contains a function definition and a variable named `num`. In ❷ you just made the function call `print(odd_or_even(num))`. You create a new scope. Notice that the global scope still exists but isn't in focus right now. In the left panel of ❷, you have the parameter `num` as a variable with the value 4. In the middle panel of ❷ you're executing the line `num=num%2` inside the function call and reassign, only in the function call scope, the variable `num` to be 0. In the right panel of ❷ you make the decision and return `"even"`. In ❸ the function returned `"even"` and the scope of the function call disappeared. You're back in the global scope and perform the two prints. Printing the function call shows `"even"` and printing the `num` shows 4 because you're using the `num` variable from the global

Find answers on the fly, or master something new. Subscribe today. See pricing options.



22.3. NESTING FUNCTIONS

Just as you can have nested loops, you can have nested functions. These are function definitions inside other functions. Python knows only about an inner function inside the scope of the outer function—and only when the outer function is being called.

Listing 22.7 shows a nested function `stop()` inside the function `sing()`. The global scope is the main program scope. It has one function definition for `sing()`. The function definition at this point is just some code. The function doesn't execute until you make a function call. When you try to say `stop()` in the main program scope, you get an error.

Inside the definition of `sing()`, you define another function named `stop()` that also contains code. You don't care what that code is until you make a function call. Inside `sing()`, the `stop()` call doesn't cause an error because `stop()` is defined inside `sing()`. As far as the main program is concerned, only function `sing` is its scope.

Listing 22.7. Nesting functions

```
def sing():
    def stop(line):                          1
        print("STOP",line)
    stop("it's hammer time")                 2
    stop("in the name of love")              2
    stop("hey, what's that sound")           2
stop()                                       3
sing()
```

- ***1* Function definition inside sing()**
- ***2* Calls inside sing() to the stop function**
- ***3* Error because stop() doesn't exist in the global scope**

Quick check 22.2

For the following code, what will each line print?

```
def add_one(a, b):
    x = a+1
    y = b+1
    def mult(a,b):
        return a*b
    return mult(x,y)

a = 1
b = 2
x = 5
y = 6
```

1

```
print(add_one(x, y))
```

2

```
print(add_one(a, b))
```


Find answers on the fly, or master something new. Subscribe today. See pricing options.

```
print(add_one(x, a))
```

4

```
print(add_one(y, b))
```

22.4. PASSING FUNCTIONS AS PARAMETERS

You've seen objects of type int, string, float, and Boolean. In Python, everything is an object, so any function you define is an object of type function. Any object can be passed around as a parameter to a function, even other functions!

You want to write code to make one of two sandwiches. A BLT sandwich tells you it has bacon, lettuce, and tomato in it. A breakfast sandwich tells you it has egg and cheese in it. In listing 22.8, `blt` and `breakfast` are both functions that return a string.

A function `sandwich` takes in a parameter named `kind_of_sandwich`. This parameter is a function object. Inside the `sandwich` function, you can call `kind_of_sandwich` as usual by adding parentheses after it.

When you call the `sandwich` function, you call it with a function object as a parameter. You give it the name of a function for the sandwich you want to make. You don't put parentheses after `blt` or `breakfast` as the argument because you want to pass the function object itself. If you use `blt()` or `breakfast()`, this will be a string object because this is a function call that returns a string

Listing 22.8. Passing a function object as a parameter to another function

```
def sandwich(kind_of_sandwich):                    1
   print("--------")
   print(kind_of_sandwich ())                      2
   print("--------")

def blt():
    my_blt = " bacon\nlettuce\n tomato"
    return my_blt

def breakfast():
    my_ec = " eggegg\n cheese"
    return my_ec

print(sandwich(blt))                               3
```

- ***1* kind_of_sandwich is a parameter.**
- ***2* kind_of_sandwich with parentheses indicates a function call.**
- ***3* Uses the function name only (the object)**

Quick check 22.3

Q1:

Draw a trace of the program in listing 22.8. At each line, decide the scope, what's printed, the variables and their values, and what the function returns, if anything.

22.5. RETURNING A FUNCTION

Because a function is an object, you can also have functions that return other functions. This is useful when you want to have specialized functions. You typically return functions when you have nested functions. To return a function object, you return only the function name. Recall that putting parentheses after the function name makes a function call, which you don't want to do.

Returning a function is useful when you want to have specialized functions inside other functions. In listing 22.9, you have a function named `grumpy`, and it prints a message. Inside the function `grumpy`, you have another function named `no_n_times`. It prints a message, and then inside that function you define another function, named `no_m_more_times`. The innermost function `no_m_more_times` prints a message and then prints `no n + m` times.

You're using the fact that `no_m_more_times` is nested inside `no_n_times` and therefore knows about the variable `n`, without having to send that variable in as a parameter.

Find answers on the fly, or master something new. Subscribe today. See pricing options.

When you make a function call with `grumpy()(4)(2)`, you work from left to right and replace function calls as you go with whatever they return. Notice the following:

- You don't have to print the return of `grumpy` because you're printing things inside the functions.
- A function call to `grumpy()` gets replaced with whatever `grumpy` returns, which is the function `no_n_times`.
- Now `no_n_times(4)` is replaced with whatever it returns, which is the function `no_m_more_times`.
- Finally, `no_m_more_times(2)` is the last function call, which prints out all the nos.

Listing 22.9. Returning a function object from another function

```
def grumpy():
    print("I am a grumpy cat:")
    def no_n_times(n):
        print("No", n,"times...")
        def no_m_more_times(m):
            print("...and no", m,"more times")
            for i in range(n+m):
                print("no")
        return no_m_more_times
    return no_n_times

grumpy()(4)(2)
```

- ***1* Function definition**
- ***2* Nested function definition**
- ***3* Nested function definition**
- ***4* Loop to print the word "no" n + m times**
- ***5* Function no_n_times returns the function no_m_more_times**
- ***6* Function grumpy returns the function no_n_times**
- ***7* Function call in main program**

This example shows that the function call is left-associative, so you replace the calls left to right with the functions they return. For example, you use `f()()()()` if you have four nested functions, each returning a function.

Quick check 22.4

Q1:

Draw a trace of the program in listing 22.9. At each line, decide the scope, what gets printed, the variables and their values, and what the function returns, if anything.

22.6. SUMMARY

In this lesson, my objective was to teach you about the subtleties of functions. These ideas only begin to scratch the surface of what you can do with functions. You created functions that had variables with the same name and saw that they didn't interfere with each other because of function scopes. You learned that functions are Python objects and that they can be passed in as parameters to or returned by other functions. Here are the major takeaways:

- You've been using built-in functions already, and now you understand why you wrote them in that way. They take parameters and return a value after doing a computation.
- You can nest functions by defining them inside other functions. The nested functions exist only in the scope of the enclosing function.
- You can pass around function objects like any other object. You can use them as parameters, and you can return them.

Let's see if you got this...

Q22.1

Fill the missing parts to the following code:

```
def area(shape, n):
    # write a line to return the area
    # of a generic shape with a parameter of n

def circle(radius):
    return 3.14*radius**2
```


Find answers on the fly, or master something new. Subscribe today. See pricing options.

```
print(area(circle,5))  # example function call
```

1. Write a line to use `area()` to find the area of a circle with a radius of 10.
2. Write a line to use `area()` to find the area of a square with sides of length 5.
3. Write a line to use `area()` to find the area of a circle with diameter of length 4.

Q22.2

Fill the missing parts to the following code:

```
def person(age):
    print("I am a person")
    def student(major):
        print("I like learning")
        def vacation(place):
            print("But I need to take breaks")
            print(age,"|",major,"|",place)
        # write a line to return the appropriate f
    # write a line to return the appropriate funct
```

For example, the function call

```
person(12)("Math")("beach")  # example function ca
```

should print this:

```
I am a person
I like learning
But I need to take breaks
12 | Math | beach
```

1. Write a function call with age of `29`, major of `"CS"`, and vacation place of `"Japan"`.
2. Write a function call so that the last line of its printout is as follows: `23 | "Law" | "Florida"`

Recommended / Playlists / History / Topics / Settings / Get the App / Sign Out

PREV
Lesson 21. Achieving modularity and abstraction with functi…

NEXT
Lesson 23. Capstone project: analyze your friends

Find answers on the fly, or master something new. Subscribe today. See pricing options.