Module 8 - Overview

Introduction

Module Learning Outcomes

After successful completion of this module, you be able to do the following:

1. Write a generator function.
2. Write a generator expression.
3. Write and use decorator functions.

Key questions:

* What is lazy evaluation, and why can it be useful?
* What's the difference between a generator expression and a list comprehension (in syntax and in behavior)?
* What does it mean that in Python, functions are "first-class objects"?
* What is an inner function?

Explorations

Use the pages within this module to explore the following concepts:

* Exploration: [Generators](https://canvas.oregonstate.edu/courses/1915078/pages/exploration-generators) (MLOs 1-2)
* Exploration: [First-class functions and decorators](https://canvas.oregonstate.edu/courses/1915078/pages/exploration-first-class-functions-and-decorators) (MLO 3)
* Video Demo: [Generators and decorators](https://canvas.oregonstate.edu/courses/1915078/pages/video-demo-generators-and-decorators) (MLOs 1-3)
* [Module 8 exercise solutions](https://canvas.oregonstate.edu/courses/1915078/pages/module-8-exercise-solutions)

Optional Resources

* [*Think Python* Chapter 19 Section 3Links to an external site.](http://greenteapress.com/thinkpython2/html/thinkpython2020.html#sec225)

Task List

Complete the following assignments and other tasks:

* Sign up for one of the [Assignment 7 - Group Part](https://canvas.oregonstate.edu/courses/1915078/assignments/9227008) groups and do the group assignment (CLO 2). Note that the groups will be available *after* the late due date for Assignment 7.
* Read the Exploration pages (linked above) and do the interactive exercises on those pages (MLOs 1-3).
* Complete [Assignment 8](https://canvas.oregonstate.edu/courses/1915078/assignments/9227009), which gives you practice with writing and using generators, first-class functions, and decorators (MLOs 1-3).
* Take [Quiz 8](https://canvas.oregonstate.edu/courses/1915078/quizzes/2859159) (MLOs 1-3).

Exploration: Generators

**Generator functions** create generator objects. A generator object produces an iterable sequence of values. In case it's not fresh in your memory, "iterable" means that we can loop through (iterate over) it using a for loop. Here's an example of a generator function to produce all the even integers from zero up to, but not including, some specified limit.

This looks like a normal function except that it uses **yield** instead of return. If we call this function, it doesn't return a number, like you might expect. Instead it returns a generator object for us to use. We could create the generator and use it like this:

A screenshot of a computer

Description automatically generated with medium confidence

This example prints out the even numbers from zero to 100. The generator function does not calculate all of those values and store them in the generator object. Instead, a generator uses **lazy evaluation**, which means that the next value is not computed until it's needed. When we need a value, the function runs until it hits yield, and then gives us that value. At that point the function is paused until we need another value. When we ask for the next value, the function picks up where it left off, and runs until it hits yield again, giving us the next value. Unlike a function, a generator remembers its state (the current values of its variables) between "calls".

But why use a generator function? Why not use a list comprehension to make a list of the values we want, and then iterate through that?

evens = [num for num in range(101) if num % 2 == 0]

Lazy evaluation is a key advantage of the generator approach. With a list comprehension, all of the values are computed at once, which, depending on the number of values needed, could take up a sizable chunk of memory. A generator can even hand you values from an infinite sequence (for example, all prime numbers) or one that's ongoing (for example, a log of page requests for a web server).

We don't have to iterate through the whole generator at once in a for loop. We can iterate through one step at a time whenever we need another value. We do this by calling the next() function:

next\_val = next(gen\_1)

If you've reached the end of the generator's values, calling next() on it will raise a StopIteration exception.

We can create simple generators with a **generator expression**, which uses a syntax very similar to a list comprehension. In fact, the only difference is that it uses parentheses at the ends instead of square brackets.

evens = (num for num in range(101) if num % 2 == 0)

Here, *evens* is a generator object. In the example of a generator function, even\_numbers was the name of the function and we created a generator object by calling that function (with a parameter, because that function takes a parameter) and assigned that generator object to gen\_1. In the line of code above, we're defining the generator expression and creating the generator object in the same line. You can use the generator object referenced by squares like this, for example:

squares = (num \* num for num in range(1,101))  
for val in squares:   
    print(val)

Unlike a list, you can only iterate over a generator once. If you want to iterate through those values again, you have to create a new generator object from the generator function or generator expression.

Exercises

1. Write a generator **expression** called "squares" that yields square numbers up to (and including) 10000 (1, 4, 9, 16, 25, etc.).

Once you have written the code, you can test it using the following tests. Make sure to rename your program file as generator\_exercise\_1 so that these tests can run successfully:

[generator\_exercise\_1\_tests.py](https://canvas.oregonstate.edu/courses/1915078/files/98542015?wrap=1)[Download generator\_exercise\_1\_tests.py](https://canvas.oregonstate.edu/courses/1915078/files/98542015/download?download_frd=1)

2. Write a generator **function** called "all\_primes" that yields **all** the prime numbers.

Once you have written the code, you can test it using the following tests. Make sure to rename your program file as generator\_exercise\_2 so that these tests can run successfully:

[generator\_exercise\_2\_tests.py](https://canvas.oregonstate.edu/courses/1915078/files/98542014?wrap=1)

# Exploration: First-class functions and decorators

## First class functions

In Python, functions are "first-class objects". That means that you can assign a function to a variable, pass a function as an argument to another function, and return a function as the return value of another function. For example, assigning a function to a variable could look like this:

A screenshot of a computer program

Description automatically generated with low confidence

Remember how variables work in Python? The variables sum and add now both refer to the same function object (functions are objects, like everything else in Python), which is why we can call the function using either name.

A screenshot of a computer program

Description automatically generated with medium confidence

perform\_binary\_op takes three parameters. The first one is a function, and the other two are values that will be passed to the function. In the example above, we called perform\_binary\_op twice - first passing sum as the function, and then passing mult as the function. perform\_binary\_op calls whatever function was given to it and passes the other two values as parameters to that function. It then returns the value returned by that function call.

### **Inner functions and closures**

We can define a function inside another function. For example:

A screenshot of a computer

Description automatically generated with medium confidence

The **inner function** named double is defined inside double\_plus\_one(), and can be used there, but it doesn't exist outside that function. If we try to call it outside that function, we'll get a NameError. However, it's possible to make it accessible by returning it from the function:

A screenshot of a computer program

Description automatically generated with medium confidence

In this example, we have a function that defines an inner function and then returns that function. Notice that in the line "twice = get\_double\_func()", the parentheses mean that we are **calling** get\_double\_func(). Without parentheses, we would just be referring to the function as an object, but with the parentheses, we are calling the function, which executes its code. The return value of get\_double\_func() is the function that it defined, which we assign to the variable named twice. Now we can use that function to double values.

Don't feel intimidated - the concepts and syntax of using functions this way takes some practice to get used to.

So we **can** define inner functions, but **why** would we do that? After all, in the double\_plus\_one example, we could have defined double() outside of the double\_plus\_one() function, and double\_plus\_one() still could have called it. And in the last example, we could have just defined double() directly, and not bothered with get\_double\_func() at all.

The answer is that the main reason we would define an inner function is if we want to create a **closure**. Let's look at an example:

A screenshot of a computer

Description automatically generated with medium confidence

Unlike get\_double\_func(), the function get\_multiplier\_ by() doesn't always return the same function. Instead, the function that gets returned depends on the argument that is passed to get\_multiplier\_by().

**Notice something odd here:** normally, when a function has finished running and returned a value, any local variables it had are no longer available - they just go away. However, in this example, that's not the case. Even after get\_multiplier\_by() has finished running and returned a function, the returned function still has access to the variable factor from get\_multiplier\_by().

A closure is an inner function that has "captured" one or more of the outer function's local variables. The value of any captured variables is accessible to the closure for as long as the closure exists. Closures are used in decorators (the next topic), but you'll also run into them in CS 290: Web Development in connection with callbacks for event handlers.

## Decorators

Now that you know more about what you can do with functions, you're ready to learn about decorators. **Decorators** allow you to give additional behavior to a function without modifying the function itself. Let's look at a generic example:

A screen shot of a computer

Description automatically generated with medium confidence

First, we define a decorator function. It takes as a parameter the function that we want to "decorate" (add behavior to). Next it defines an inner "wrapper" function that wraps up the original function together with the additional behavior we want. Then it returns that wrapper function (it doesn't have to be named "wrapper", but it's usually fine to do so). After defining the decorator function, we define the function we're going to decorate, which just prints "hello world". Next we assign the decorated version of the function to the name of the function. Now, whenever we use that name, it refers to the decorated version. When we call greeting(), it no longer just prints "hello world". It now also performs the (generic) additional behavior we gave it.

There's a special syntax in Python that would replace the line that makes the function assignment. Using that syntax for the example above gives us:

A screen shot of a computer

Description automatically generated with medium confidence

The line that made the function assignment is gone, and instead we have the line "@generic\_decorator" just before the definition of the function we're decorating.

What if the function we're decorating has a return value? As the wrapper function is now, any return value from the function we're decorating would be lost. If greeting returned "hello world" instead of printing it out, like this:

@generic\_decorator  
def greeting():  
 """Returns a greeting"""  
 return "hello world"  
  
print(greeting())

Then we would expect the print statement at the bottom to print "hello world". The call to func in the wrapper function returns that string, but the wrapper function doesn't pass it along. We can amend that like so:

A screenshot of a computer

Description automatically generated with medium confidence

Now the wrapper function stores the return value of the function we're decorating and returns it, after it finishes running all of the additional behavior we defined. Now the print statement at the bottom does in fact print "hello world".

What if the function we're decorating takes arguments? We could make the wrapper function also take arguments and then pass those arguments in the function call, but the problem is that we might want to use the same decorator for different functions that take different numbers of arguments. Python has us covered. It provides a way to pass a variable number of arguments using **\*args** and **\*\*kwargs**.

The wrapper function now take a variable number of arguments using \*args and \*\*kwargs and passes them on to the function that's being decorated:

A screenshot of a computer program

Description automatically generated with medium confidence

Now we can decorate a function that takes any number of arguments, like in:

print(increment(3))

Or like this:

print(add\_three(2,3,4))

Every function we decorate using "@generic\_decorator" now has the additional behavior we defined in that decorator function, and it works for any function, regardless of return value or number of arguments.

There is one more subtlety to address, which is that the wrapper function hides the metadata of the decorated function. Metadata are certain things an object knows about its own properties. For example, you've seen that we can use the "help" command to get an object's documentation (provided in its docstring). If we call "help(add\_three)", without add\_three being decorated, we get:

add\_three(num\_1, num\_2, num\_3)  
 Adds three numbers

But if we do it for add\_three with decoration, we get:

wrapper(\*args, \*\*kwargs)

The wrapper is obscuring the add\_three function's metadata. Perhaps fittingly, the way to fix this is by using a decorator that Python provides. That decorator is defined in the functools module so you need to import that. Specifically, we'll use it to decorate the wrapper function:

A screenshot of a computer program

Description automatically generated with low confidence

Now if we call "help" for the decorated add\_three function, we get what we want.

It's best practice to use functools.wraps() to decorate the wrapper in all your decorator functions.

The only behavior added by the decorator example we've been working with is a couple of generic print statements. Let's look at an example that prints out the values of all a function's parameters.

A screenshot of a computer program

Description automatically generated with medium confidence

We didn't have to change much from our generic version. Before the decorated function is called, we loop through args, printing out the value of each argument. We don't need anything to happen after the decorated function is called.

In general decorators are very useful for adding behavior to a function that isn't really part of its primary purpose, and which you might want to apply to a number of different functions. Other examples include printing information about a function for debugging purposes, making authentication checks, doing input validation, etc. It's helpful to be able to add the extra behavior, but keep the definition of that behavior separate from the functions' definitions, since copy-pasting the code for the additional behavior into a bunch of different functions would clutter up their definitions and also be rather tedious. And if you want to update the added behavior, you only need to do it in one place instead of across many functions.

## Exercises

1. Write a function named greeting, that takes someone's name as a parameter and returns a string consisting of "Hello " followed by that name. For example, greeting("Jemma") should return (not print out) "Hello Jemma". Next write a function named shout that will take that function and someone's name as parameters, call the function on the name to get the greeting string, convert the greeting string to all-caps, and return the capitalized greeting. For example, shout(greeting, "Jemma") should return (not print out) "HELLO JEMMA".

Once you have written the code, you can test it using the following tests. Make sure to rename your program file as **first\_class\_exercises.py** so that these tests can run successfully:

[first\_class\_functions\_test\_1.py](https://canvas.oregonstate.edu/courses/1915078/files/98542016?wrap=1)[Download first\_class\_functions\_test\_1.py](https://canvas.oregonstate.edu/courses/1915078/files/98542016/download?download_frd=1)

2. Write a decorator named debug that prints "Entering " followed by the name of the decorated function (func.\_\_name\_\_) before calling it and prints "Exiting " followed by the name of the decorated function after calling it.

You will have to test this out yourself.

<https://media.oregonstate.edu/media/t/1_0e2iwswr>