# Intro to Programming: Functions

In this lesson, you will learn how to organize your code with functions. A function is a block of code designed to perform a specific task. As you'll see, functions will let you do roughly the same calculation multiple times without duplicating any code.

## Intro to Functions: A Simple Example

We begin with a simple example of a function. The add\_three() function below accepts any number, adds three to it, and then returns the result.

# Define the function

def add\_three(input\_var):

output\_var = input\_var + 3

return output\_var

Every function is composed of two pieces: a header and a body.

### Header

The function header defines the name of the function and its argument(s).

* Every function header begins with def, which tells Python that we are about to define a function.
* In the example, the function name is add\_three.
* In the example, the argument is input\_var. The argument is the name of the variable that will be used as input to the function. It is always enclosed in parentheses that appear immediately after the name of the function. (Note that a function can also have no arguments, or it can have multiple arguments. You'll see some examples of this later in the lesson.)
* For every function, the parentheses enclosing the function argument(s) must be followed by a colon :.

### Body

The function body specifies the work that the function does.

* Every line of code in the function body must be indented exactly four spaces. You can do this by pushing the space bar four times, or by hitting the "Tab" button once on your keyboard. (As you learn more about Python, you may need to indent your code by more than four spaces, but you'll learn more about that later in this course.)
* The function does its work by running all of the indented lines from top to bottom.
  + It takes the argument as input, which in the example is input\_var.
  + The function creates a new variable output\_var with the calculation output\_var = input\_var + 3.
  + Then, the final line of code, called the **return statement**, just returns the value in output\_var as the function's output.

The code cell above just defines the function, but does not run it. The details of the function body will make more sense after the next code cell, when we actually run the function.

## How to Run (or "Call") a Function

When we run a function, it can also be referred to as "calling" the function.

In the code cell below, we run the function with 10 as the input value. We define a new variable new\_number which is set to the output of the function.

# Run the function with 10 as input

new\_number = add\_three(10)

# Check that the value is 13, as expected

print(new\_number)

**Output:**

13

In more detail:

* add\_three(10) is the value that we get as output when we supply 10 as the value for input\_var and call the add\_three() function. When the function runs, it runs all of the code in its body, from top to bottom:
  + It first calculates output\_var = input\_var + 3, which sets output\_var = 13.
  + The final line of code is the **return statement**, which returns the value of output\_var, which is 13.
* By setting new\_number = add\_three(10), we set new\_number = 13.

**Note:** When we casually refer to the add\_three() function in this tutorial, we use empty closing parentheses after the function name. This is consistent with how people generally write explanations of Python code, and the empty parentheses just make it clear that we are referring to a function, as opposed to a variable or another Python object. These parentheses should always be empty, even if the function has arguments.

## Naming Functions

In the example above, the name of the function was selected for you. When naming your own functions, you should use only lowercase letters, with words separated by underscores instead of spaces.

Naming functions will feel natural over time, and it is normal for it to feel uncomfortable at first. The best way to learn is by viewing a lot of Python code.

## A More Complex Example

Now that you understand the basics, we can move on to an example with a longer calculation.

Say you are helping a friend to calculate their weekly paycheck after taxes.

* They're in a 12% tax bracket (in other words, 12% of their salary is taken for taxes, and they only take home 88%), and
* They're paid hourly, at a rate of $15/hour.

The function below calculates the paycheck based on the number of hours worked. The function is more complicated than with the first example, because the function has more lines of code and comments. Similar to the example above, the function has a single argument (num\_hours). In the function body, we:

* Use the value for num\_hours to specify the value for a new variable pay\_pretax.
* Use the value of pay\_pretax to specify the value for a new variable pay\_aftertax.
* Return the value of the pay\_aftertax variable.

def get\_pay(num\_hours):

# Pre-tax pay, based on receiving $15/hour

pay\_pretax = num\_hours \* 15

# After-tax pay, based on being in 12% tax bracket

pay\_aftertax = pay\_pretax \* (1 - .12)

return pay\_aftertax

We call this function the same way we called the first function. The next code cell calculates the paycheck, based on working 40 hours. (After taxes, it is $528.)

# Calculate pay based on working 40 hours

pay\_fulltime = get\_pay(40)

print(pay\_fulltime)

**Output:**

528.0

To quickly calculate pay based on a different number of hours worked, you need to supply the function with a different number. For instance, say your friend works 32 hours. (Then, they get $422.40.)

pay\_parttime = get\_pay(32)

print(pay\_parttime)

**Output:**

422.4

Because you wrote a function, you can calculate pay for different hours without having to rewrite all of the code in the calculations all over again.

Functions can help you to avoid errors in your code, and you save a lot of time. In general, when coding, you should aim to write as little as possible, because each time you type out a calculation, it's another opportunity to accidentally introduce a typo or error.

## Variable "Scope"

Variables defined inside the function body cannot be accessed outside of the function. For instance, the next code cell errors, because pay\_aftertax only exists inside the function.

print(pay\_aftertax)

**Output:**

---------------------------------------------------------------------------

NameError Traceback (most recent call last)

/tmp/ipykernel\_19/2991332200.py in <module>

----> 1 print(pay\_aftertax)

NameError: name 'pay\_aftertax' is not defined

You will get the same error if you try to print pay\_pretax or num\_hours. For this reason, if you need any information from a function, you need to make sure that appears in the return statement at the end of the function.

We refer to a variable's **scope** as the part of the code where it is accessible. Variables defined inside a function (like pay\_aftertax) have a **local scope** of that function only. However, as you've seen, variables defined outside all functions (like pay\_parttime) have a **global scope** and can be accessed anywhere.

## Functions with Multiple Arguments

So far, you have learned how to define a function with just one argument. To define a function with multiple arguments, you only need to add more arguments inside the parentheses in the function head and separate them with a comma.

We do this with the get\_pay\_with\_more\_inputs() function below, which calculates a weekly paycheck based on three arguments:

* num\_hours - number of hours worked in one week
* hourly\_wage - the hourly wage (in $/hour)
* tax\_bracket - percentage of your salary that is removed for taxes

def get\_pay\_with\_more\_inputs(num\_hours, hourly\_wage, tax\_bracket):

# Pre-tax pay

pay\_pretax = num\_hours \* hourly\_wage

# After-tax pay

pay\_aftertax = pay\_pretax \* (1 - tax\_bracket)

return pay\_aftertax

Then, to call the function, you need to provide one value for each input, again separated by a comma.

In the code cell below, we calculate the pay after taxes for someone who works 40 hours, makes $24/hour, and is in a 22% tax bracket.

higher\_pay\_aftertax = get\_pay\_with\_more\_inputs(40, 24, .22)

print(higher\_pay\_aftertax)

**Output:**

748.8000000000001

The following code cell gives the same result as when we ran get\_pay(40), because hourly\_wage is set to 15, and tax\_bracket is set to 12%, which lines up with how we designed get\_pay.

same\_pay\_fulltime = get\_pay\_with\_more\_inputs(40, 15, .12)

print(same\_pay\_fulltime)

**Output:**

528.0

Depending on how we plan to use this new function get\_pay\_with\_more\_inputs(), it can be more useful than the original function get\_pay(), because it addresses more cases. Instead of potentially incorrectly assuming the hourly wage and tax bracket, the new function allows the user to specify the correct values. But, if you're sure the hourly wage and tax bracket won't need to change, the new function is just more complicated than necessary. In general, when defining functions, you'll need to consider how much flexibility you need, based on your use case.

## Functions with No Arguments

Note that it's possible to define a function with no arguments, and that doesn't have a return statement. The print\_hello() function in the code cell below is an example.

# Define the function with no arguments and with no return

def print\_hello():

print("Hello, you!")

print("Good morning!")

# Call the function

print\_hello()

**Output:**

Hello, you!

Good morning!

This course is short and meant as an introduction to programming, so we won't cover everything you need to know about functions in this tutorial. You'll learn even more about functions in the Python course.