Lesson 5: Functions

In programming, as you start to write longer and more complex programs, one thing you'll start to notice is how often the same set of steps need to be repeated in many different places within a program.

A **function** is a key element in writing such programs. You can think of a function in a computing language in much the same way you think of a mathematical function. The function takes in arguments, performs some operation based on the identities of the arguments, and then returns a result. For example, the mathematical function *f(x,y)=xy* takes arguments *x* and *y* and then returns the product of those two arguements, *xy*. In this lesson, we will learn how to construct functions in Python.

> **Note:** Some tasks need to be performed multiple times within a program. Rather than rewrite the same code in multiple places, a function may be defined using the `def` keyword. Function definitions may include **parameters**, providing data input to the function.
>
> Functions may return a value using the `return` keyword followed by the value to return.

Sometimes functions require input to provide data for their code. This input is defined using the **function parameters** mentioned previously. Parameters are variables that are defined in the function definition. They are assigned the values which were passed as arguments when the function was called, elsewhere in the code.

> **Multiple Parameters:** Python functions can have multiple parameters. Just as you wouldn't go to school without both a backpack and a pencil case, functions may also need more than one input to carry out their operations. To define a function with multiple parameters, parameter names are placed one after another, separated by commas, within the parentheses of the function definition.

For example, the function definition defines parameters for a character, a setting, and a skill, which are used as inputs to write the first sentence of a book.

Basic function syntax

For the above example, let's translate the above function into Python. A function is defined using the `def` keyword. This is best seen by example:

In [1]:
```
def product(x, y):
    """The product of `x` and `y`."""
    return x * y
```

Following the `def` keyword is a *function signature* which indicates the function's name and its arguments. Just like in mathematics, the arguments are separated by commas and enclosed in parentheses. The indentation following the `def` line specifies what is part of the function. As soon as the indentation goes to the left again, aligned with `def`, the contents of the functions are complete.

Immediately following the function definition is the *doc string* (this is short for documentation string), which is a brief description of the function. The first string after the function definition is always defined as the doc string. Usually, it is in triple quotes, as doc strings can often span multiple lines.

Doc strings are more than just comments for code, the doc string is what is returned by the native python function `help()` when someone is looking to learn more about a function.

In [2]:
```
help(product)
```

```
Help on function product in module __main__:

product(x, y)
    The product of `x` and `y`.
```

> They are also printed out when the `?` is used in a Jupyter notebook or JupyterLab console.

In [3]:
```
product?
```

```
Signature: product(x, y)
Docstring: The product of `x` and `y`.
File:      c:\users\ayla\appdata\local\temp\ipykernel_18340\853791213.py
Type:      function
```

You're free to put whatever you like in doc strings, or even omit them, but you should always have a doc string with some information about what your function is doing. True, this example of a function is kind of silly, since it is easier to type `x * y` than `product(x, y)`, but it is still good form to have a doc string. It's good documentation and communication, and every programmer should build up the habit sooner rather than later.

It's worth saying explicitly: ***All functions should have doc strings.***

In the next line of the function, there's the `return` keyword. Whatever is after the `return` statement is (obviously) returned by the function. (More about this later.) Any code after the `return` isn't executed because the function has already returned.

Calling a function

Now that our function has been defined, we can call it.

Python uses simple syntax to use, invoke, or call a preexisting function. A function can be called by

writing the name of it, followed by parentheses. For example, the code provided would call the `product()` method.

In [4]:
```
product(2,5)
```

Out[4]: 10

In [5]:
```
product(3,6)
```

Out[5]: 18

In [6]:
```
product(0.5,8)
```

Out[6]: 4.0

In [7]:
```
product(-1,6)
```

Out[7]: -6

Note the cases where a `float` is returned, despite only one parameter being a float because that's how multiplication works.

Function Arguments

Parameters in python are variables: placeholders for the actual values the function needs. When the function is called, these values are passed in as arguments. For example, the arguments passed into the function `.sales()` are the "The Farmer's Market", "toothpaste", and "$1" which correspond to the parameters `grocery_store`, `item_on_sale`, and `cost`.

In [8]:
```
def sales(grocery_store, item_on_sale, cost):
  print(grocery_store + " is selling " + item_on_sale + " for " + cost)

sales("The Farmer's Market", "toothpaste", "$1")
```

```
The Farmer's Market is selling toothpaste for $1
```

Function Keyword Arguments

Python functions can be defined with named arguments which may have default values provided. When function arguments are passed using their names, they are referred to as *keyword arguments*. The use of keyword arguments when calling a function allows the arguments to be passed in any order — not just the order that they were defined in the function. If the function is invoked without a value for a specific argument, the default value will be used.

In [9]:
```
def findvolume(length=1, width=1, depth=1):
  print("Length = " + str(length))
  print("Width = " + str(width))
  print("Depth = " + str(depth))
  print("\t Volume = " + str(length*width*depth))
  return length * width * depth;
```

```
findvolume(1, 2, 3)
findvolume(length=5, depth=2, width=4)
findvolume(2, depth=3, width=4)
```

```
Length = 1
Width = 2
Depth = 3
         Volume = 6
Length = 5
Width = 4
Depth = 2
         Volume = 40
Length = 2
Width = 4
Depth = 3
         Volume = 24
```

Out[9]: 24

Functions Without Arguments

A function doesn't necessarily need arguments. As a silly example, let's consider a function that just returns `42` every single time it's called. Of course, it does not matter what its arguments are, so we can define a function without arguments.

In [10]:
```
def answer_to_the_ultimate_question_of_life_the_universe_and_everything():
    """Simpler program than Deep Thought's, I bet."""
    return 42
```

We still needed the open and closed parentheses at the end of the function name. Similarly, even though it has no arguments, we still have to call it with parentheses.

In [11]:
```
answer_to_the_ultimate_question_of_life_the_universe_and_everything()
```

Out[11]: 42

Returning Value from Function

As mentioned previously, a `return` keyword is used to return a value from a Python function. The value returned from a function can be assigned to a variable which can then be used in the program.

In the example, the functi on `check_leap_y` ear returns a string which indicates if the passed parameter is a leap year or not.

In [12]:
```
def check_leap_year(year):
  if year % 4 == 0:
    return str(year) + " is a leap year."
  else:
    return str(year) + " is not a leap year."

year_to_check = 2023
```

```
returned_value = check_leap_year(year_to_check)
print(returned_value) # 2023 is not a leap year.
```

```
2023 is not a leap year.
```

In [ ]:
```
print(check_leap_year(2024))
```

Functions need not return anything

Hwoever, just like they do not necessarily need arguments, functions also do not need to return anything. If a function does not have a `return` statement (or if it's never encountered in the execution of the function), the function runs to completion and returns `None` by default. `None` is a special Python keyword which basically means "nothing." For example, a function could simply print something to the screen.

In [13]:
```
def think_too_much():
    """Express Caesar's skepticism about Cassius"""
    print("""Yond Cassius has a lean and hungry look,
He thinks too much; such men are dangerous.""")
```

Let's call this function like the others, but we can show that the result it returns is `None`.

In [14]:
```
return_val = think_too_much()
```

```
Yond Cassius has a lean and hungry look,
He thinks too much; such men are dangerous.
```

In [ ]:
```
# Print the return value
print(return_val)
```

All About Variables

To relate back to a previous lesson, let's bring back the topic of *variables* for a moment. These are programming symbols/words (e.g. `x` and `y` and `vegetable`) that can be assigned and hold information, whether `int`, `float`, or `string`. In relation to functions, there are two kinds of variables: "local" and "global".

The Scope of Variables

In Python, a *variable* defined inside a function is called a *local variable*. It cannot be used outside of the scope of the function, and attempting to do so without defining the variable outside of the function will cause an error.

In the example, the variable `a` is defined both inside and outside of the function. When the function `f1()` is implemented, `a` is printed as `2` because it is locally defined to be so. However, when printing a outside of the function, `a` is printed as 5 because it is implemented outside of the scope of the function.

In [15]:
```
a = 5
```

```
def f1():
  a = 2
  print(a)
```

In [ ]:
```
print(a)   # Will print 5
```

In [ ]:
```
f1()    # Will print 2
```

Parameters as Local Variables

Function parameters behave identically to a function's local variables. They are initialized with the values passed into the function when it was called.

Like local variables, parameters cannot be referenced from outside the scope of the function.

In the example, the parameter value is defined as part of the definitio `n of my_fun` ction, and therefore can only be accessed wi `thin my_fun` ction. Attempting to print the contents of value from outside the function causes an error.

In [16]:
```
def my_function(value):
  print(value)

# Pass the value 7 into the function
my_function(7)
```

```
7
```

In [ ]:
```
# Causes an error as `value` no longer exists
print(value)
```

Global Variables

A variable that is defined outside of a function is called a *global variable*. It can be accessed inside the body of a function.

In the example, the variable `a` is a global variable because it is defined outside of the function `prints_a`. It is therefore accessible to `prints_a`, which prints the value of `a`.

In [ ]:
```
a = "Hello" # global variable

def prints_a():
  print(a)

# will print "Hello"
prints_a()
```

Built-in functions in Python

The Python programming language has several built-in functions. We've already discussed

`print()`, `id()`, `ord()`, `len()`, `range()`, `enumerate()`, `zip()`, and `reversed()`, in addition to type conversions such as `list()`. The complete set of built-in functions can be found here. A word of warning about these functions and naming your own: ***Never define a function or variable with the same name as a built-in function.***

Additionally, Python has keywords (such as `def`, `for`, `in`, `if`, `True`, `None`, etc.), many of which have already been encountered. A complete list of them is here. The interpreter will throw an error if you try to define a function or variable with the same name as a keyword.

A Final Note

Let's pause for a moment and think about what functions do. Not just in the abstract sense of "oh, what is a function and what is its place within my program", but also in the specific sense of "what do I want *this particular* function to do?" And, more to the point: how do we expect a certain function to act? Every time you sit down to write a function (or really any program in general, it's just good practice), ask yourself these questions:

- What is this code supposed to do?
- And how can I make sure it's doing that?

The answer to that second question is to write a test. A *function test* is a set of parameter(s) that we call a function with such that the returned value(s) is known. For example, taking our previous `product()` function and using the age-old rule "anything multiplied by 0 is 0", we can write the test:

In [ ]:
```
product(0,1)
```

As expected, it produces `0`. Obviously, this is a very simple test, but the idea is to come up with a simple, easy way of testing your function where you already know the answer.

For now, we'll proceed without writing tests, but I want you to keep this in mind. For now, just know this: If your functions are not thoroughly tested, you are entering a world of pain. **A world of pain.**

Test your functions.

Proceed to Assignment 5.