Introduction to Computer Programming

Week 4.2: Function arguments and scope

Arguments

- Arguments allow input to be provided to functions
- When defining a function, multiple arguments must be separated by commas
- When we call a function, the order of the arguments we provide must match the order of arguments the function expects
- This is why docstrings are helpful!

Why does the argument order matter?

Let's consider a function that prints a name:

In [1]:
```
def print_name(first_name, last_name):
    print(first_name, last_name)
```

If this function is called using `print_name(Matt, Hennessy)`, then:

1. The argument *first_name* is assigned the value of 'Matt'
2. The argument *last_name* is assigned the value of 'Hennessy'
3. The values of *first_name* and *last_name* are printed
4. The function terminates

By default, the arguments in a function are assigned values in the order they are provided

Keyword arguments

Python, in fact, allows arguments to be assigned in any order using **keyword arguments**.

The idea is to provide the names of the arguments as well as their values when calling a function.

For example

```
my_function(a = 1.0, b = 2.0, c = 2.3)
```

Example: Consider the `print_name` function

In [2]:
```
def print_name(first_name, last_name):
    print(first_name, last_name)
```

We'll call this function using standard (positional) arguments and then keyword arguments

In [1]:
```
# call using positional arguments (order matters)
```

In [2]:
```
# call using keyword arguments (order does not matter)
```

Default arguments

If a function argument usually takes on the same value, then we can assign a default value to it.

- This means that this argument does not need to have a value passed to it when the function is called.
- It also means that this argument must be **optional**

To create a default argument, the default value of the argument is assigned in the function definition

```
def fun(default_argument = default_value):
```

Example: Let's return to the name-printing function. Now, let's add an option for the names to be printed in reverse order by passing a Boolean called *reverse*:

In [3]:
```
def print_name(first_name, second_name, reverse):
    if reverse:
        print(second_name + ', ' + first_name)
    else:
        print(first_name, second_name)
```

Now let's use a default argument to automatically set the *reverse* parameter equal to False.

In [6]:
```
def print_name(first_name, second_name, reverse = False):
    if reverse:
        print(second_name + ', ' + first_name)
    else:
        print(first_name, second_name)
```

This means we can call the function without providing a third argument:

In [ ]:
```

```

However, if we do want to use reverse order, then we can pass the third argument to override the default value:

In [ ]:
```

```

Variable number of arguments

Python enables functions with a variable number of arguments to be written using the **unpacking operator** *

Example: Write a function that sums an arbitrary number of numbers. Each number will be passed to the function as an argument.

In [ ]:
```

```

How does the unpacking operator work?

The unpacking operator * tells the function to create a tuple out of the arguments it receives.

To see this, let's consider the `my_sum` function. Now we'll add a line that prints the type of *numbers*

In [10]:
```
def my_sum(*numbers):
    print(type(numbers))
    S = 0
    for n in numbers:
        S += n
    return S

S = my_sum(1, 2, 3, 4, 5)
```

```
<class 'tuple'>
```

Variable scope

The variables defined in a function only exist within that function and cannot be accessed outside of it

- These variables are said to be in the **local scope** of the function
- This is why we need to use the `return` keyword to produce outputs

Example: Accessing a local variable *c* outside of the function in which it is defined triggers an error

In [4]:
```
# this function adds a and b, saves the result in c
```

The **scope** of a variable describes the part of the program in which it can be accessed

Example: Consider the two functions

In [12]:
```
def fun1():
    x = 1

def fun2():
    y = 2
```

The scope of *x* is `fun1` and the scope of *y* is `fun2`. Both *x* and *y* have local scope.

Variables that are defined in main body of Python code have **global scope**. They can be accessed anywhere, even within functions we define!

Example: Use global scope to print the value of a variable *x* in a function with no arguments

In [ ]:
```

```

In [ ]:
```

```

The code runs without error. Since *x* has global scope, it can be accessed by the `print_x` function

The `global` keyword

A variable with local scope can obtain global scope using the `global` keyword

In [ ]:
```

```

Even though *c* was defined in the `add` function, it has global scope, so it can be accessed anywhere

Good programming practice - global variables

Just because something can be done, doesn't mean it should be done

Global variables should be avoided whenever possible. Since they can be accessed and modified anywhere in the program, it becomes difficult to keep track of these changes and find mistakes if they occur!

The exercises will demonstrate these points

Summary

- By default, arguments are assigned values in the order they are provided
- Keyword arguments can be used to assign values to arguments in any order
- Default arguments pre-assign values to optional arguments
- The scope of a variable describes where a variable can be accessed
- Variables with global scope can be accessed anywhere, but should be avoided when possible