Functions with Arguments

Introduction

Function arguments are a powerful tool in programming. As we'll see, arguments make our functions more flexible and reusable by explicitly allowing different inputs to be used in a function and changing the output of the function based on these inputs.

When used correctly, function arguments bring clarity to what inputs a function needs to operate, as well as how a function uses these inputs.

Objectives

You will be able to:

- Declare and use a function with arguments

Predictability with arguments

In the previous lesson, we saw that functions were a powerful tool. They allow us to repeat operations and apply these operations to different data. For example, take a look at a function called `meet_traveller()` .

In [1]:
```
def meet_traveller():
    welcome_message = "Hi " + traveller.title() + ", I'm so glad we'll be going c
    return welcome_message # return statement
```

The `meet_traveller()` function is designed to generate nice greetings to each new employee. Do we need anything else to run the function? How do we know which new employee the function will greet? Let's run the function and see what happens.

In [2]:
```
meet_traveller()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipykernel_475/2590242057.py in <module>
----> 1 meet_traveller()

/tmp/ipykernel_475/3942520332.py in meet_traveller()
      1 def meet_traveller():
----> 2     welcome_message = "Hi " + traveller.title() + ", I'm so glad we'll
 be going on the trip together!"
      3     return welcome_message # return statement

NameError: name 'traveller' is not defined
```

The function requires the variable `traveller` , but it's hard to tell that before running the function.

When code requires something to work, we call that something a **dependency**. Below, our function `meet()` depends on a global variable named `traveller` being provided, otherwise it will not work and we'll get the `NameError` we see above. Ideally, our function's dependencies would be more explicit than in our `meet_traveller()` function. Let's adapt this function to make its dependencies more explicit.

In [3]:
```
def meet(traveller):
    welcome_message = "Hi " + traveller.title() + ", I'm so glad we'll be going o
    return welcome_message
```

Ok, in the code above we changed the first line of the function, the function signature, to the following:

```
def meet(traveller):
```

This tells us, and Python, to not even run the code unless the proper data is provided to our function. Let's see what this means by calling the function **without** its argument.

In [4]:
```
meet()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_475/1098287491.py in <module>
----> 1 meet()

TypeError: meet() missing 1 required positional argument: 'traveller'
```

Do you see that error message at the bottom there? It's pretty explicit about saying that this function requires a positional argument named `traveller`.

So, by using an argument, the function signature tells us how to run this function. We refer to the function by its name and then pass through a string representing the `traveller`.

In [5]:
```
meet('sally')
```

Out[5]:
```
"Hi Sally, I'm so glad we'll be going on the trip together!"
```

> **Note:** Before we move on, it is important to note the difference between an **argument** and a **parameter**. We will see these terms **a lot** going forward and having a clear understanding of them is imperative. An **argument** is the data that we pass to a function before execution (i.e. `'sally'` in the example above). However, a **parameter** is the variable we define in the *function signature* (i.e. `traveller` in the example above). So, these two are very similar, but the difference comes down to whether we are talking about the variable we are using in the function *signature* or the actual data we are using when we *execute* the function later.

Flexibility

Let's take another look at the `meet()` function. Notice, that the argument operates like a variable in that we can easily alter the data that `traveller` points to. When we pass through the string, `'Sally'`, the function replaces `traveller` with the string `'Sally'`.

In [5]:
```
def meet(traveller):
                                # "sally"
    welcome_message = "Hi " + traveller.title() + ", I'm so glad we'll be going o
    return welcome_message
```

And we can easily change what `traveller` points to just by passing through a different string.

In [6]:
```
meet('fred')
```

Out[6]:
```
"Hi Fred, I'm so glad we'll be going on the trip together!"
```

But notice that the `traveller` argument is only accessible just inside of the function.

In [7]:
```
traveller
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipykernel_475/2902578661.py in <module>
----> 1 traveller

NameError: name 'traveller' is not defined
```

So by using arguments, we can easily see what a function requires to work, change the output by passing through different data to the function, and ensure that we only have to worry about what our argument is while inside that function.

Now, we can use functions with arguments to do a lot more than just make some strings more dynamic. Let's say we have a math operation that we need to perform over and over. Let's say, the mean of a list of numbers. We could define a function named `find_the_mean()` that takes in a list and returns a number representing the mean from the list.

In [8]:
```
def find_the_mean(list_nums):
    length = len(list_nums)
    total = sum(list_nums)
    return total/length
```

In [9]:
```
find_the_mean([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

Out[9]:
```
5.5
```

Now let's imagine we are looking at a list of populations in a given state or region. Perhaps we would even like to get these numbers in order to compare the mean populations of different areas.

In [10]:
```
area_one_pops = [10453, 12383, 8034, 800835, 76434, 32394]
find_the_mean(area_one_pops)
```

Out[10]:
```
156755.5
```

In [11]:
```
area_two_pops = [43845, 54930, 59354, 96403, 73492, 729320]
find_the_mean(area_two_pops)
```

Out[11]:
```
176224.0
```

Great, so, we can *definitely* find the mean from lists of population. What if we would like to return the list that has the largest mean population? We could write a function that takes **2** arguments that are lists and returns the list that has the greatest mean.

In [12]:
```
def find_biggest_pop(list_pops_one, list_pops_two):
    mean_one = find_the_mean(list_pops_one)
    mean_two = find_the_mean(list_pops_two)
    if (mean_one > mean_two):
        return print("The first list,", list_pops_one ," has the larger mean popu
    else:
        return print("The second list,", list_pops_two ,"has the larger mean popu
```

In [13]:
```
find_biggest_pop(area_one_pops, area_two_pops)
```

```
The second list, [43845, 54930, 59354, 96403, 73492, 729320] has the larger mea
n population
```

Awesome! Going forward, we will be able to use functions and arguments to write code that is much more reusable and concise to model mathematical and statistical equations.

Summary

In this lesson, we saw some of the benefits of using arguments: they make our functions more flexible and predictable. Our functions are more flexible as the functions vary based on the argument provided to the function. Arguments make our functions predictable by making functions more explicit about their dependencies. They also allow us to change the value of an argument which only affects the function's internal values and more directly shows us how the output of our function will vary based on different inputs.