Functions

Functions will be one of our main building blocks when we construct larger and larger amounts of code to solve problems.

So what is a function?

Formally, a function is a useful device that groups together a set of statements so they can be run more than once. They can also let us specify parameters that can serve as inputs to the functions.

On a more fundamental level, functions allow us to not have to repeatedly write the same code again and again. If you remember back to the lessons on strings and lists, remember that we used a function `len()` to get the length of a string. Since checking the length of a sequence is a common task you would want to write a function that can do this repeatedly at command.

Functions will be one of the most basic levels of **reusing code in Python**.

`def` Statements

Let's see how to build out a function's syntax in Python. It has the following form:

```
def example_function(arg1, arg2):
  '''
  This is where the function's Document String (docstring) goes
  '''
  # Do stuff here
  return desired_result
```

- We begin with `def` then a space followed by the name of the function. Try to keep names relevant, for example `len()` is a good name for a `length()` function. Also be careful with names, you wouldn't want to call a function the same name as a built-in function in Python (such as len).
- Next come a pair of parentheses with a number of arguments separated by a comma. These arguments are the inputs for your function. You'll be able to use these inputs in your function and reference them. After this you put a colon.
- Now here is the important step, you must indent to begin the code inside your function correctly. Python makes use of *whitespace* to organize code. Lots of other programing languages do not do this, so keep that in mind.
- Next you'll see the docstring, this is where you write a basic description of the function. Using iPython and iPython Notebooks, you'll be able to read these docstrings by pressing Shift+Tab after a function name. Docstrings are not necessary for simple functions, but it's good practice to put them in so you or other people can easily understand the code you write.

After all this you begin writing the code you wish to execute.

- The best way to learn functions is by going through examples. So let's try to go through examples that relate back to the various objects and data structures we learned about before.
- `return` allows a function to *return* a result that can then be stored as a variable, or used in whatever manner a user wants.

Example 1: Extract a list of first letters from a sentence

```
heisenberg_quote = "It ceases to exist without me. No, you clearly don't know who you're talking to, so let me clue you in. I am not in dang
```

```
# Using the list comprehension, we can do the above task in this way
words_by_walter = heisenberg_quote.split(' ')

first_letters = [word[0] for word in words_by_walter]
print(first_letters)
```

```
['I', 'c', 't', 'e', 'w', 'm', 'N', 'y', 'c', 'd', 'k', 'w', 'y', 't', 't', 's', 'l', 'm', 'c', 'y', 'i', 'I', 'a', 'n', 'i', 'd', 'S',
```

Now let us write a function which does the same task for any given sentence

```
def extract_first_letters(sentence):
  '''
```

```
  This function takes a sentence as an input and returns
  the list of first letters of each word
  '''

  # Step 1 : Get the list of words in this sentence
  words_in_sentence = sentence.split(' ')

  # Step 2: Write a list comprehension to extract the first letters
  first_letters = [word[0] for word in words_in_sentence]

  # Step 3: Return the output list first_letters
  return first_letters


# Store the result of the function in another variable
first_letters_heisenberg = extract_first_letters(heisenberg_quote)
print(first_letters_heisenberg)
```

```
['I', 'c', 't', 'e', 'w', 'm', 'N', 'y', 'c', 'd', 'k', 'w', 'y', 't', 't', 's', 'l', 'm', 'c', 'y', 'i', 'I', 'a', 'n', 'i', 'd', 'S',
```

Let us test this function on another sentence

```
starwars_quote = "May the Force be with you."
first_letters_starwars = extract_first_letters(starwars_quote)
print(first_letters_starwars)
```

```
['M', 't', 'F', 'b', 'w', 'y']
```

Example 2 : Let us now write a function which returns the factorial of a number.

In mathematics, the factorial of a positive integer n, denoted by $n!$, is the product of all positive integers less than or equal to n

```
def factorial(num):
    '''
    Function for calculating factorial of a number
    '''

    # Step 1 : Write an if  statement to print a message if the input number is negative
    if num < 0:
      print("Factorial is not defined for negative numbers.")

    # Step 2: Write the elif statement to calculate the factorial
    elif num > 0:
      result = 1
      for i in range(1, num+1):
        result = result * i
      # print(f"The factorial of {num} is {result}.")

    # Step 3: Write the else statement to take care of the input 0
    else:
      print("The factorial of 0 is 1.")

    # Step 4 : Return the output variable result

    return result


fact_seven = factorial(5)
print(fact_seven)
```

```
120
```

```
# Now let us define a list of numbers for which we want to calculate the factorials
list_of_numbers = [3, 5, 8, 12, 4, 6]

# We can write a list comprehension for the same task
factorial_list = [factorial(number) for number in list_of_numbers if number%2!=0]
print(factorial_list)
```

```
[6, 120]
```

Example 3: Let us write a function which takes a date as a string and prints the quarter as output.

Remember it is not required that functions will return a value. This is an example of such functions

```
def assignQuarter(user_date):
  '''
  This function prints the quarter for a given date string
  '''

  # Step 1 : Extract the year and month from the input date string
  date_year = user_date[:4]
  date_month = user_date[5:7]

  # Step 2 : Store the quarter value in a string variable named quarter
  if (date_month >= '01') & (date_month <='03'):
    quarter = date_year + '-Q1'
  elif (date_month >= '04') & (date_month <='06'):
    quarter = date_year + '-Q2'
  elif (date_month >= '07') & (date_month <='09'):
    quarter = date_year + '-Q3'
  else:
    quarter = date_year + '-Q4'

  # Step 3 : Print the message you want to display to the user. Here we are not returning anything
  print(f'The corresponding quarter for the date {user_date} is {quarter}')
  # return quarter


sample_date = '2020-07-01'
date_quarter = assignQuarter(sample_date)
print(date_quarter)
```

```
The corresponding quarter for the date 2020-07-01 is 2020-Q3
None
```

```
my_quarter = assignQuarter(sample_date)
```

```
The corresponding quarter for the date 2020-07-01 is 2020-Q3
```

```
print(my_quarter)
```

```
None
```

```
assignQuarter
```

Example 4 : Let us write a function with multiple return statements

```
def check_even_or_odd(num):
  '''
  This function checks whether a number is odd or even and then returns the corresponding string
  '''
  print(f'Given number is {num}')

  # Step 1 : Check if the number is even. If it is return 'even'
  if num%2 == 0:
    return 'even'

  # Step 2 : Else just return 'odd'

  return 'odd'


is_even_odd = check_even_or_odd(10)
print(is_even_odd)
```

```
Given number is 10
even
```

Example 5 : Next let us write a function which returns multiple variables at once.

In this specific example, the input is a list of numbers. And we have to return the mean and median of this list of numbers

```
heights =  [172,175,170,168,170,200]


def calculate_averages(my_list):
  '''
  This function calculates the mean and median of a given list of numbers.
  '''
  # Step 1 : Calculate the mean
  mean = sum(my_list)/len(my_list)

  # Step 2 : Calculate the median
  sorted_list = sorted(my_list)
  list_length = len(sorted_list)

  if list_length%2 == 0:
    median = (sorted_list[int(list_length/2)-1] + sorted_list[int(list_length/2)])/2
  else :
    median = sorted_list[int((list_length+1)/2) - 1]

  return  mean, median


heights =  [172,175,170,168,170,200]
calculate_averages(heights)
```

```
(175.83333333333334, 171.0)
```

```
mean_height, median_height = calculate_averages(heights)


mean_height
```

```
175.83333333333334
```

```
median_height
```

```
171.0
```

Example 6: Now let us write a function which adds an element to a list. This functions takes the list as an argument and the element to be added

```
def list_append(my_list, elem):
  '''
  This function appends an element to the list
  '''
  # Step 1 : Add the element to the list
  new_list = my_list + [elem]

  # Step 2 : Return the new_list
  return new_list


print(heights)
```

```
[172, 175, 170, 168, 170, 200]
```

```
list_append(heights,190)
```

```
[172, 175, 170, 168, 170, 200, 190]
```

```
print(heights)
```

```
[172, 175, 170, 168, 170, 200]
```

```
list_append(heights, elem = 40)
```

```
[172, 175, 170, 168, 170, 200, 40]
```

```
print(heights)
```

```
[172, 175, 170, 168, 170, 200]
```

```
heights.append(60)
```

```
print(heights)
```

```
[172, 175, 170, 168, 170, 200, 60]
```

```
Start coding or generate with AI.
```

Let us find out what are the differences between functions and methods

- Function and method both look similar as they perform in an almost similar way, but the key difference is the concept of 'Class and its Object'. Functions can be called only by its name, as it is defined independently. But methods cannot be called by its name only we need to invoke the class by reference of that class in which it is defined, that is, the method is defined within a class and hence they are dependent on that class.
- A method is called by its name but it is associated with an object (dependent). It is implicitly passed to an object on which it is invoked. It may or may not return any data. A method can operate the data (instance variables) that is contained by the corresponding class.

Methods

We've already seen a few example of methods when learning about lists in Python. Methods are essentially functions built into objects.

Methods perform specific actions on an object and can also take arguments, just like a function.

Methods are of the form:

```
object.method(arg1,arg2,etc...)
```

```
def insert_value_to_list(list_obj, index_to_place, value_to_place):
  '''
  This function inserts an element to a list at a specified index
  '''

  # Step 1 : Get the first part
  first_part = list_obj[0:index_to_place]

  # Step 2 : Append the element at the end of the first part
  first_part.append(value_to_place)

  # Step 3 : Get the last part
  last_part = list_obj[index_to_place:]

  # Step 4 : Use the extend method to add the last part to the first part
  first_part.extend(last_part)

  # Step 5 : Return the first part
  return first_part
```

```
new_heights = insert_value_to_list(heights, 2, 180)
```

```
new_heights
```

```
[172, 175, 180, 170, 168, 170, 200]
```

```
heights.insert(2,180)
```

```
heights
```

```
[172, 175, 180, 170, 168, 170, 200]
```

Scope

Now that we have gone over writing our own functions, it's important to understand how Python deals with the variable names you assign. When you create a variable name in Python the name is stored in a *name-space*. Variable names also have a *scope*, the scope determines the visibility of that variable name to other parts of your code.

Let's start with a quick thought experiment; imagine the following code:

```
def experiment():
  global x
  x = 50

  return x

x = 25
```

What do you imagine the output of `experiment()` is? 25 or 50? What is the output of print `x`? 25 or 50?

```
experiment()
```

```
50
```

```
experiment()
```

```
50
```

```
x
```

```
50
```

How do we make this function take the global variable `x` and change it?

```
def new_experiment():

  x = 50

  return x

x = 40
```

```
x
```

```
40
```

```
new_experiment()
```

```
50
```

```
Start coding or generate with AI.
```