Function and Methods in Python

Why we need functions?

- Creating clean repeatable code is a key part of becoming an effective programmer.
- **Functions** allow us to create blocks of code that can be easily executed many times, without needing to constantly rewrite the entire block of code.

Pre-defined built-in functions in Python

- Python has a number of built-in functions. For example:
  - `print()`, `input()`, `type()`, `help()`, `dir()`
  - `len()`, `sum()`, `min()`, `max()`
  - `str()`, `int()`, `float()`, etc.

Pre-defined functions in Python objects

- Python's built-in objects such as, `str`, `list`, `dict`, etc. have their own pre-defined functions (also known as methods).
- For example, some of the pre-defined methods for the str object are:
  - `.lower()`, `.upper()`, `.strip()`, `.startswith()`, `.endswith()`, and many more
  - You can see all the pre-defined methods for the str object by running
    - `dir(str)` -> this returns a list of the names of all pre-defined methods for the `str`
  - The `list` object has methods like: `.append()`, `.reverse()`, `.sort()`

Example of calling Python’s built-in functions and object methods

- You can use Python’s built-in functions simply by calling with name and proving the required arguments. For example,
  - message = input(‘Enter a message: ’)
- To call an object’s method, example:
  - `my_list = [1, 2, 3]`
  - `my_list.reverse()# Calling the reverse() method on our list object`
  - `“hello”.upper()  # Calling the upper() method on our string object`

Writing your own Functions

def Keyword

Creating your own functions

- Creating a function requires a very specific syntax, including the **def** keyword, correct indentation, and proper structure.
- Let’s get an overview of a Python function structure.

The def keyword

```
def name_of_function():
```

Keyword telling Python this is a function.

Naming the function

```
def name_of_function():
```

You decide on the function name. Notice “snake casing”

Naming the function

```
def name_of_function():
```

Snake casing is all lowercase with underscores between words

```
def name_of_function():
```

Parenthesis at the end. Later on we can pass in arguments/parameters into the function.

Starting the function body

```
def name_of_function():
```

A colon indicates an upcoming indented block. Everything indented is then “inside” the function

Writing function's documentation (optional)

```
def name_of_function():
    '''
    Docstring explains function.
    '''
```

Optional: Multi-line string to describe function.

Use proper indentation inside function body

```
def name_of_function():
    '''
    Docstring explains function.
    '''
```

Note: Everything inside the function is indented

Writing the body of the function

```
def name_of_function():
    '''
    Docstring explains function.
    '''
    print("Hello")
```

Code then goes inside the function.

Calling the function

```
def name_of_function():
    '''
    Docstring explains function.
    '''
    print("Hello")
```

```
>> name_of_function()
>> Hello
```

Function can then be executed/called to see the result.

Calling the function

```
def name_of_function():
    '''
    Docstring explains function.
    '''
    print("Hello")
```

```
>> name_of_function()
>> Hello
```

Resulting Output

Functions with parameters / arguments

```
def name_of_function(name):
    '''
    Docstring explains function.
    '''
    print("Hello "+name)
```

```
>> name_of_function("Jose")
>> Hello Jose
```

Functions can accept arguments to be passed by the user.

Functions with parameters / agruments

```
def name_of_function(name):
    '''
    Docstring explains function.
    '''
    print("Hello "+name)
```

```
>> name_of_function("Jose")
>> Hello Jose
```

Functions can accept arguments to be passed by the user.

Returning something (data) from a function

- Typically we use the **return** keyword to send back the result of the function, instead of just printing it out.
- **return** allows us to assign the output of the function to a new variable.

Returning from function

```
def add_function(num1,num2):
    return num1+num2
```

Return allows to save the result to a variable.

```
>> result = add_function(1,2)
>>
>> print(result)
>> 3
```

Returning from function

```
def add_function(num1,num2):
    return num1+num2
```

Most functions will use return. Rarely will a function only print()

```
>> result = add_function(1,2)
>>
>> print(result)
>> 3
```

Putting it all together

- Let’s start creating functions with Python.
- Putting it all together in the next example.

Write a function that takes a list of numbers, and returns all the even numbers:

Return all even numbers in a list

Let's add more complexity, we now will return all the even numbers in a list, otherwise return an empty list.

In [35]:
```
def check_even_list(num_list):

    even_numbers = []

    # Go through each number
    for number in num_list:
        # Once we get a "hit" on an even number, we append the even number
        if number % 2 == 0:
            even_numbers.append(number)
        # Don't do anything if its not even
        else:
            pass
    # Notice the indentation! This ensures we run through the entire for loop
    return even_numbers
```

In [36]:
```
check_even_list([1,2,3,4,5,6])
```

Out[36]: [2, 4, 6]

In [37]:
```
check_even_list([1,3,5])
```

Out[37]: []

*args and **kwargs

`*args` and `**kwargs`

Work with Python long enough, and eventually you will encounter `*args` and `**kwargs`. These strange terms show up as parameters in function definitions. What do they do? Let's review a simple function:

In [1]:
```
def myfunc(a,b):
    return sum((a,b))*.05

myfunc(40,60)
```

Out[1]: 5.0

This function returns 5% of the sum of **a** and **b**. In this example, **a** and **b** are *positional* arguments; that is, 40 is assigned to **a** because it is the first argument, and 60 to **b**. Notice also that to work with multiple positional arguments in the `sum()` function we had to pass them in as a tuple.

What if we want to work with more than two numbers? One way would be to assign a *lot* of parameters, and give each one a default value.

In [2]:
```
def myfunc(a=0,b=0,c=0,d=0,e=0):
    return sum((a,b,c,d,e))*.05

myfunc(40,60,20)
```

Out[2]: 6.0

Obviously this is not a very efficient solution, and that's where `*args` comes in.

Using *args

`*args`

When a function parameter starts with an asterisk, it allows for an *arbitrary number* of arguments, and the function takes them in as a tuple of values. Rewriting the above function:

In [3]:
```
def myfunc(*args):
    return sum(args)*.05

myfunc(40,60,20)
```

Out[3]: 6.0

Notice how passing the keyword "args" into the `sum()` function did the same thing as a tuple of arguments.

It is worth noting that the word "args" is itself arbitrary - any word will do so long as it's preceded by an asterisk. To demonstrate this:

In [4]:
```
def myfunc(*spam):
    return sum(spam)*.05

myfunc(40,60,20)
```

Out[4]: 6.0

Using **kwargs

`**kwargs`

Similarly, Python offers a way to handle arbitrary numbers of *keyworded* arguments. Instead of creating a tuple of values, `**kwargs` builds a dictionary of key/value pairs. For example:

In [1]:

```
def myfunc(**kwargs):
    if 'fruit' in kwargs:
        print(f"My favorite fruit is {kwargs['fruit']}")  # review String Formatting and f-strings if this syntax is unfamil
    else:
        print("I don't like fruit")

myfunc(fruit='pineapple')
```

```
My favorite fruit is pineapple
```

In [6]:

```
myfunc()
```

```
I don't like fruit
```

Using both *args and **kwargs in a function

`*args` and `**kwargs` combined

You can pass `*args` and `**kwargs` into the same function, but `*args` have to appear before `**kwargs`

In [7]:
```
def myfunc(*args, **kwargs):
    if 'fruit' and 'juice' in kwargs:
        print(f"I like {' and '.join(args)} and my favorite fruit is {kwargs['fruit']}")
        print(f"May I have some {kwargs['juice']} juice?")
    else:
        pass

myfunc('eggs','spam',fruit='cherries',juice='orange')
```

```
I like eggs and spam and my favorite fruit is cherries
May I have some orange juice?
```

Placing keyworded arguments ahead of positional arguments raises an exception:

In [8]:
```
myfunc(fruit='cherries',juice='orange','eggs','spam')
```

```
  File "<ipython-input-8-fc6ff65addcc>", line 1
    myfunc(fruit='cherries',juice='orange','eggs','spam')
                                          ^
SyntaxError: positional argument follows keyword argument
```

As with "args", you can use any name you'd like for keyworded arguments - "kwargs" is just a popular convention.

That's it! Now you should understand how `*args` and `**kwargs` provide the flexibilty to work with arbitrary numbers of arguments!

Questions?