😊 Functions in Python

A function is a **named block of code** designed to perform a specific task. It allows you to **reuse code** instead of writing the same logic multiple times. Functions can take **inputs (parameters)**, process them, and return **outputs (results)**.

Use Cases of Functions:

1. **Performing calculations** – A function can add, subtract, or analyze data.
2. **Processing data** – It can filter, sort, or manipulate information.
3. **Automating tasks** – Repetitive actions like sending emails or generating reports can be automated.
4. **Handling user input** – Functions can validate and process user-provided information.

Benefits of Using Functions:

- **Code Reusability** – Write once, use multiple times.
- **Improved Readability** – Makes complex programs easier to understand.
- **Easier Debugging** – Issues can be fixed in one place instead of multiple occurrences.
- **Better Organization** – Large programs are structured into smaller, manageable parts.

😄 To begin with the Lab

1. This defines a function that takes two inputs (**arg1, arg2**). The **docstring** explains what the function does. Inside, it performs some actions (**Do stuff here**) and then **returns** a result. Using help(function_name), you can see the docstring for guidance.

[1]:
```
def name_of_function(arg1,arg2):
    '''
    This is where the function's Document String (docstring) goes.
    When you call help() on your function it will be printed out.
    '''
    # Do stuff here
    # Return desired result
```

2. We defined a function **say_hello()**, and we will run it as you see below in the snapshot it will return us hello on the screen. It doesn't take any input or return anything, just displays a greeting.
3. Also, when we called the function, we used parenthesis and if we do not use them then the output would be like this.

[2]:
```
def say_hello():
    print('hello')
```

[4]:
```
say_hello()
```

```
hello
```

[6]:
```
say_hello
```

[6]:
```
<function __main__.say_hello()>
```

4. Now we will write a function that will greet people by their name.

[8]:
```
def greeting(name):
    print(f'Hello {name}')
```

[10]:
```
greeting('Tom and Jerry')
```

```
Hello Tom and Jerry
```

5. So far we've only seen **print()** used, but if we actually want to save the resulting variable we need to use the **return** keyword.
6. We have defined a function to add numbers and then return the added value as output. After that, we saw that we can also save the result as a variable due to return.

[12]:
```
def add_num(num1,num2):
    return num1+num2
```

[14]:
```
add_num(4,5)
```

[14]:
```
9
```

[16]:
```
# Can also save as variable due to return
result = add_num(4,5)
```

[18]:
```
print(result)
```

```
9
```

7. In **cell [20]**, the function add_num('one', 'two') was called with **strings** instead of numbers. Since the + operator is used, Python **concatenates** (joins) the two strings instead of adding numbers. So, 'one' + 'two' results in 'onetwo', which is why that output appears.

```
[20]: add_num('one','two')

[20]: 'onetwo'
```

8. So, let us understand the difference between return and print. **The return keyword allows you to actually save the result of the output of a function as a variable. The print() function simply displays the output to you, but doesn't save it for future use.**
9. Here we used both of the function print and return to get the output.

```
[22]: def print_result(a,b):
          print(a+b)

[24]: def return_result(a,b):
          return a+b

[26]: print_result(10,5)

      15

[28]: # You won't see any output if you run this in a .py script
      return_result(10,5)

[28]: 15
```

10. But what happens if we really want to save the result? In the example below, we have used print result to get the output but if you see here, the result is not saved.

```
[5]: my_result = print_result(20,20)

     40

[6]: my_result

     ...

[7]: type(my_result)

[7]: NoneType
```

11. Now we have used the return function to get the output and it is actually saved in the memory.

```
[8]: my_result = return_result(20,20)
     •••
[9]: my_result
[9]: 40
[10]: my_result + my_result
[10]: 80
```

12. Below, we are just checking if the number is even or not. If you remember **the mod operator %, which returns the remainder after division, if a number is even then mod 2 (% 2) should be == to zero.**

```
[11]: 2 % 2
[11]: 0
[12]: 20 % 2
[12]: 0
[14]: 21 % 2
[14]: 1
[15]: 20 % 2 == 0
[15]: True
[16]: 21 % 2 == 0
[16]: False
```

13. Now let's use this to construct a function. Notice how we simply return the Boolean check.

[18]:
```
def even_check(number):
    return number % 2 == 0
```

[19]:
```
even_check(20)
```

[19]:
```
True
```

[21]:
```
even_check(21)
```

[21]:
```
False
```

14. Let's return a Boolean indicating if **any** number in a list is even. Notice here how **return** breaks out of the loop and exits the function
15. This function **checks if a list has any even numbers**.
    It goes through each number in num_list.
    If it finds an **even number** (divisible by 2), it **returns True immediately**.
    If it doesn't find any even numbers, it **does nothing** (pass just means "skip").
16. **Problem:** If the list has no even numbers, the function returns **nothing (None)** because there's no return False statement.

[25]:
```
def check_even_list(num_list):
    # Go through each number
    for number in num_list:
        # Once we get a "hit" on an even number, we return True
        if number % 2 == 0:
            return True
        # Otherwise we don't do anything
        else:
            pass
```

17. Here in the first call, it finds 1 and it is an odd number, so it returns nothing and moves on, but then it finds 2 and it is an even number, so it immediately returns true.
18. But in the second call, it cannot find any even number but still did not return any result.
19. The issue is that the function should return **False** if no even number is found. Right now, it only returns True for even numbers but nothing otherwise.

[26]:
```
check_even_list([1,2,3])
```

[26]:
```
True
```

[27]:
```
check_even_list([1,1,1])
```

20. Now, if you look at this function, the **return False statement is inside the loop**, which causes a problem.

[28]:
```
def check_even_list(num_list):
    # Go through each number
    for number in num_list:
        # Once we get a "hit" on an even number, we return True
        if number % 2 == 0:
            return True
        # This is WRONG! This returns False at the very first odd number!
        # It doesn't end up checking the other numbers in the list!
        else:
            return False
```

[30]:
```
# UH OH! It is returning False after hitting the first 1
check_even_list([1,2,3])
```

[30]:
```
False
```

21. Here, you can see that we have used the correct approach, where this function **checks if a list contains any even numbers**.
22. It loops through each number in the list if it finds an **even number** (number % 2 == 0), it **immediately returns True**. If it doesn't find any even number, it **returns False after checking all numbers**.

[31]:
```
def check_even_list(num_list):
    # Go through each number
    for number in num_list:
        # Once we get a "hit" on an even number, we return True
        if number % 2 == 0:
            return True
        # Don't do anything if its not even
        else:
            pass
    # Notice the indentation! This ensures we run through the entire for loop
    return False
```

[32]:
```
check_even_list([1,2,3])
```

[32]:
```
True
```

[34]:
```
check_even_list([1,3,5])
```

[34]:
```
False
```

23. In the example below, we are returning only the even number; otherwise return an empty list.

[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
```

...

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

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

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

[37]:
```
[]
```

24. We can also loop through the list of tuples and unpack the values within them.

[38]:
```
stock_prices = [('AAPL',200),('GOOG',300),('MSFT',400)]
```

•••

[39]:
```
for item in stock_prices:
    print(item)
```

```
('AAPL', 200)
('GOOG', 300)
('MSFT', 400)
```

[41]:
```
for stock,price in stock_prices:
    print(stock)
```

```
AAPL
GOOG
MSFT
```

[42]:
```
for stock,price in stock_prices:
    print(price)
```

```
200
300
400
```

25. Similarly, functions often return tuples to easily return multiple results for later use.

[46]:
```
work_hours = [('Abby',100),('Billy',400),('Cassie',800)]
```

The employee of the month function will return both the name and number of hours worked for the top performer (judged by number of hours worked).

[47]:
```
def employee_check(work_hours):

    # Set some max value to intially beat, like zero hours
    current_max = 0
    # Set some empty value before the loop
    employee_of_month = ''

    for employee,hours in work_hours:
        if hours > current_max:
            current_max = hours
            employee_of_month = employee
        else:
            pass

    # Notice the indentation here
    return (employee_of_month,current_max)
```

[48]:
```
employee_check(work_hours)
```

[48]:
```
('Cassie', 800)
```

26. Now we will look at the interaction between functions. Functions often use results from other functions.
27. Let's see a simple example through a guessing game. There will be 3 positions in the list, one of which is an **'O'**, a function will shuffle the list, another will take a player's

guess, and finally, another will check to see if it is correct. This is based on the classic carnival game of guessing which cup a red ball is under.

[8]:
```
example = [1,2,3,4,5]
```

•••

[9]:
```
from random import shuffle
```

•••

[10]:
```
# Note shuffle is in-place
shuffle(example)
```

[11]:
```
example
```

[11]:
```
[3, 1, 4, 5, 2]
```

[12]:
```
mylist = [' ','O',' ']
```

[13]:
```
def shuffle_list(mylist):
    # Take in list, and returned shuffle versioned
    shuffle(mylist)

    return mylist
```

[14]:
```
mylist
```

[14]:
```
[' ', 'O', ' ']
```

[15]:
```
shuffle_list(mylist)
```

[15]:
```
[' ', ' ', 'O']
```

[18]:
```
def player_guess():

    guess = ''

    while guess not in ['0','1','2']:

        # Recall input() returns a string
        guess = input("Pick a number: 0, 1, or 2:  ")

    return int(guess)
```

[24]:
```
player_guess()
```

```
Pick a number: 0, 1, or 2:  1
```

[24]:
```
1
```

Now we will check the user's guess. Notice we only print here, since we have no need to save a user's guess or the shuffled list.

[22]:
```
def check_guess(mylist,guess):
    if mylist[guess] == '0':
        print('Correct Guess!')
    else:
        print('Wrong! Better luck next time')
        print(mylist)
```

Now we create a little setup logic to run all the functions. Notice how they interact with each other!

[23]:
```
# Initial List
mylist = [' ','0',' ']

# Shuffle It
mixedup_list = shuffle_list(mylist)

# Get User's Guess
guess = player_guess()

# Check User's Guess
#------------------------
# Notice how this function takes in the input
# based on the output of other functions!
check_guess(mixedup_list,guess)
```

```
Pick a number: 0, 1, or 2:  1
Wrong! Better luck next time
[' ', ' ', '0']
```