

**Defining Functions**:

So far we have learned, how to use built in functions in python, like print, round and so on.

In this section we will learn how to write our own functions.

***“****The reason of writing functions is to break down our code into smaller, maintainable and potentially more reusable chunks, these chunks are called functions****”***.

🡪 We use *def* keyword to define a function, followed by function name, parentheses and a colon.

def greet():

🡪 Below the *greet* function we get indentation where we write our statements.

def greet():

    print("Hello there")

    print("Welcome aboard")

Note: Make sure your custom function names are meaningful, descriptive, use *lowercase* letters and *separate multiple words in function name with an underscore*.

🡪 At last we *call* the *greet* function.

greet()

**Arguments**:

While defining a function in between parentheses we write our parameters.

def greet(first\_name, last\_name):

    print("Hello there")

    print("Welcome aboard")

Here we add two parameters first\_name and last\_name, so while calling this function we need to *supply two values for those parameters*. Those values we refer to as *arguments*.

greet("Himanshu", "Pandey")

These are arguments for the *greet* function.

***Difference between parameter and argument***:

A parameter is the input that we define for our function while arguments is the actual value for a given parameter.

Now let us implement these parameters in our functional block

def greet(first\_name, last\_name):

    print(f"Hello there {first\_name} {last\_name}")

    print("Welcome aboard")

greet("Himanshu", "Pandey")

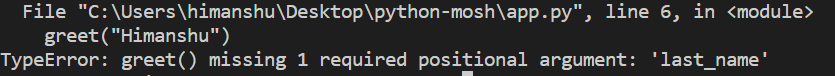
O/P: 

Now we can use different arguments for different outputs which makes this code more reusable.

Note: By default all the parameters that we define for a function are required. Here our greet function requires two parameters, so if I remove one argument. I get type error.

greet("Himanshu")

error:



**Types of functions**:

In programming we have two types of functions,

🡪 Functions that *perform a task* like *print* or *greet* function which prints a message on terminal.

🡪 Functions that calculate and *return a value* like *round* function which calculate a value and returns it.

We can convert below function which falls in first category

def greet(name):

    print(f"Hi {name}")

to a function in second category which returns a value.

Here we simply used **return** keyword for returning a value from the function.

def get\_greeting(name):

    return f"Hi {name}"

Since this function returns a value, we can store that value in a variable,

message = get\_greeting("Himanshu")

Which implementation is better *greet* or *get\_greeting*?

A> with *greet* function we are simply printing a statement on the terminal, if tomorrow we have to write the same statement in file or send as an email to all users, we have to write a separate function.

In contrast , second form *get\_greeting* is not tied to same functionality, *it simply returns a value. What we do with that value is up to us*.

Note: consider this function,

def greet(name):

    print(f"Hi {name}")

if we print this function,

print(greet("Himanshu"))

we will get , None.



None is the return value of the *greet* function. In python all functions by default return None value(*unless we specifically return a value*). *None is an object that represents absence of a value*.

**Keyword Arguments**:

Let us create another function called increment where we will increment a number by a given value.

def increment(number, by):

    return number + by

Now print the result,

result = increment(2, 1)

print(result)

O/P: 3

Since we are using result variable only in a single place, we can simplify this,

def increment(number, by):

    return number + by

print(increment(2, 1))

and we get the same results because when python interpreter executes this code , first it will call *increment* function, get the result and *temporarily store in a variable for us*(*we cannot see that variable*) and then it will pass that variable as an argument to the print function.

*How to make this code more readable*?

If someone looks at our last line,

print(increment(2, 1))

They might not know what exactly these arguments are for.

We can use a *keyword argument* to make this code more readable.

We can prefix our arguments with the name of parameters

def increment(number, by):

    return number + by

print(increment(2, by=1))

Now we can read this code almost like plain English, *Increment 2 by 1*. Here *by=1* is a *keyword argument*.

**Default Arguments**:

Earlier we saw that all the parameters that we define in our functions are required by default. In this lecture we will know *how to make parameters optional*.

def increment(number, by):

    return number + by

print(increment(2, by=1))

We will make this *by* parameter optional.

Assume we do not want to pass this *by* parameter explicitly, every time we want to call this *increment* function. It should automatically increment the value by 1 if we do not give any argument for *by* while calling *increment* function.

We can give this *by* parameter a default value like this,

def increment(number, by=1):

    return number + by

print(increment(2))

O/P: 3

But if we pass some other value for *by*,

def increment(number, by=1):

    return number + by

print(increment(2, 5))

O/P: 7

Note: Make sure all the optional parameters comes after the required parameters otherwise you will get an error in function definition.

**\*args, wait, what**?

There are times when you want to create a *function that takes a variable number of arguments*.

Let us take example of a simple multiply function which takes two parameters x and y and returns their product.

def multiply(x, y):

    return x \* y

multiply(2, 3)

While calling we can only pass two arguments but what if we want to pass more arguments…

multiply(2, 3, 4, 5) ..... and so on…

To solve this problem, we need to replace x and y parameters with a single parameter(*use plural name to indicate collection of parameters*)

def multiply(numbers):

and prefix it with an *asterisk* **\***(*the magical part*!*!*)

def multiply(\*numbers):

Let us simply print the numbers for now,

def multiply(\*numbers):

    print(numbers)

multiply(2, 3, 4, 5)

O/P: (2, 3, 4, 5)

We see all our arguments packed in parentheses. This is a *tuple* which is a *collection of objects*. The key thing about a tuple is that we cannot modify this collection(*like adding new object*).

We can iterate over a tuple which means we can use them in loops.

def multiply(\*numbers):

    for number in numbers:

        print(number)

multiply(2, 3, 4, 5)

O/P: 

To calculate product of all numbers:

def multiply(\*numbers):

    total = 1

    for number in numbers:

        total = total \* number //or total\*=number

    return total

print(multiply(2, 3, 4, 5))

O/P: 120

**\*\*args**:

In last lecture we learned syntax to pass a variable number of arguments to a function. We have a variation of this syntax with *double asterisk*(\*\*)

def multiply(\*\*numbers):

Let us assume a new function to save information about the user,

def save\_user(\*\*user):

Now print this *user* parameter in terminal,

def save\_user(\*\*user):

    print(user)

save\_user()

O/P: {}

We get an empty dictionary.

Now let us pass some *keyword arguments* in this function,

def save\_user(\*\*user):

    print(user)

save\_user(id=1, name="John", age=22)

O/P:



Notice the syntax, *between two curly braces, we have multiple key-value pairs*.

*Takeaway*:

When we pass double asterisk to function parameter, we can pass multiple key-value pairs or multiple keyword arguments to a function and python will automatically package them into a dictionary.

Note: Using [ ] syntax we can access any value from the dictionary,

def save\_user(\*\*user):

    print(user["name"])

O/P: John

**Scope**:

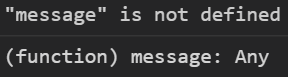
In programming we have a very important concept called *scope*, which *refers to the region of the code where the variable is defined*.

For example:

def greet():

    message="a"

In this code, Consider this *message* variable. The scope of this variable only exist inside of this function. So if you go outside *greet* function and try to print(*message*), we get a *NameError*



The same rule applies on the parameters of our function,

def greet(name):

    message = "a"

print(name)

We get the same *NameError*.

So the scope of *name* and *message* variables is the *greet* function and we refer to these variable as *local variables*. *They are local in function means they do not exists anywhere else*.

This means we can have another function,

def send\_email(name):

    message = "B"

with same parameter and variable names but completely different scope wise.

These local variables have short life time. So when we call

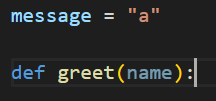
greet("Himanshu")

python interpreter will allocate some memory and have the *name* and *message* variables reference those memory locations.

*What happens to these local variables after execution is finished*?

**“***When it will finish executing the greet function, since these variables are not referenced or used anywhere else, eventually they get garbage collected. It means python interpreter will release the memory allocated for these variables****”***.

In contrast to local variables we have *global* variables. So if we move this *message* variable outside the *greet* function,



Now it is a *global variable which means it is accessible anywhere in this file*, *so the scope of this variable is the current file*.

Note: Global variables are evil! because they stay in the memory for a longer period of time until they are garbage collected and you should not use them that often. *As a best practice create functions with parameters and local variables*.

VERY BAD PRACTICE: **DO NOT DEFINE GLOBAL VARIABLE INSIDE A FUNCTION**.

**Debugging**:

This part of lecture, you can see again If needed. Try debugging on this code.

def multiply(\*numbers):

    total = 1

    for number in numbers:

        total \*= number

    return total

print("Start")

print(multiply(1, 2, 3))

O/P: 6 (*correct*)

Introduce a bug:

def multiply(\*numbers):

    total = 1

    for number in numbers:

        total \*= number

        return total

print("Start")

print(multiply(1, 2, 3))

O/P: 1(*incorrect*)

Open debugger and try to see where the issue is coming from.

**Fizz Buzz Problem**:

*Rules*:

🡪 Print “Fizz” if input divisible by 3 only

🡪 Print “Buzz” if input divisible by 5 only

🡪 Print “Fizz Buzz” if input is divisible by 3 and 5 both

🡪 Print input number if number is not divisible by 3 or 5.

This is the most elegant solution…

def fizz\_buzz(input):

    if (input % 3 == 0) and (input % 5 == 0):

        return "FizzBuzz"

    if input % 3 == 0:

        return "Fizz"

    if input % 5 == 0:

        return "buzz"

    return input

print(fizz\_buzz(15))

O/P : FizzBuzz