# Introduction to
# Python
## Programming

## UNIT 4

**Prof. Somesh Nandi**
**Dept. of AIML**
**RV College of Engineering**
**Bengaluru**

*Go, Change the World*

# Outline

→ Functions

- Creating Functions

- Using Parameters and Return Values

- Using Keyword Arguments and Default Parameters Values

- Using Global Variables and Constants

→ Files and Exceptions

- Reading from Text Files

- Writing to Text Files

- Handling Exceptions

# Functions

- **Functions**
    - A function in Python is **a block of code which only runs when it is called**.
    - You can pass data, known as parameters, into a function.
    - A function can return data as a result.

- **Need of functions**
    - Once your programs reach a certain size or level of complexity, it becomes hard to work with them this way.
    - Fortunately, there are ways to break up big programs into smaller, manageable chunks of code.
    - These manageable chunks are called as functions.

# Functions (contd.)

## Python Function Declaration

**The syntax to declare a function is:**

**def function_name(arguments):**
    **# function body**


    **return**

**Here,**


**def - keyword used to declare a function**

**function_name - any name given to the function**

**arguments - any value passed to function**

**return (optional) - returns value from a function**

## Calling a Function

**To call a function, use the function name followed by parenthesis:**

```python
def my_function(): # function definition
  print("Hello from a function")


my_function() # Function calling
```

**Eg:**
```python
def greet():
    print('Hello World!')
greet()
```

# Functions (contd.)

- **Defining Functions / Function creation**

- Function blocks begin with the keyword **def** followed by the **function-name** and **parentheses** ( ( ) ).
- Any input **parameters or arguments** should be placed within these parentheses. You can also define parameters inside these parentheses.
- The first statement of a function can be an optional statement - the documentation string of the function or docstring.
- The code block within every function starts with a **colon (:)** and is **indented**.
- The statement **return [expression]** exits a function, **optionally passing back** an expression to the caller. A return statement with no arguments is the same as return None.

**Syntax**
```
def functionname( parameters ):
    "function_docstring"
    function_suite / Body of function
    return [expression]
```

This line tells the computer that the block of code that follows is to be used together as the function instructions(). I'm basically naming this block of statements. This means that whenever I call the function instructions() in this program, the block of code runs.

- **Example**.

```
def greet():
    " This will print the hello world"
    print('Hello World!')


# call the function
greet()


print('Outside function')
```
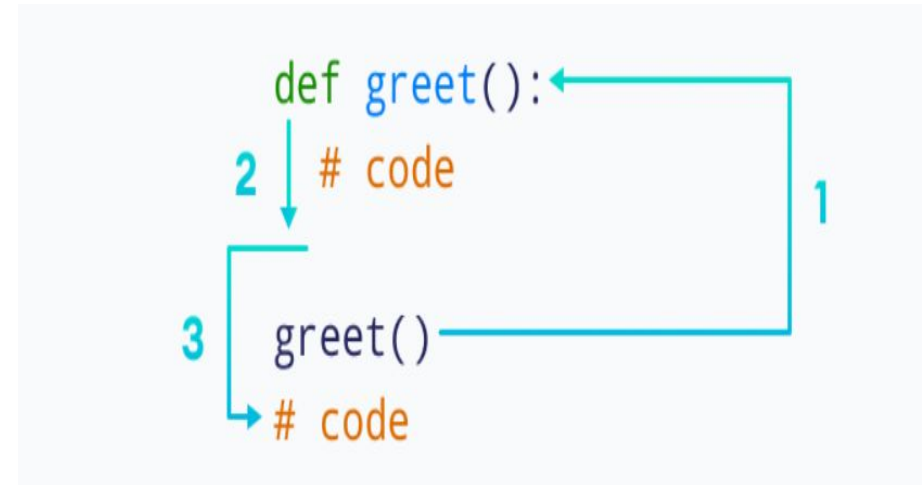
# Function Parameters/ Arguments

## Arguments /Parameters to function

- Information can be passed into functions as arguments.
- Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.

The following example has a function with one argument (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

**Example**

```
def my_function(fname):
    print(fname + " RVCE")

my_function("BT")
my_function("CD")
my_function("AIML")
```

From a function's perspective:

A parameter is the variable listed inside the parentheses in the function definition.

An argument is the value that is sent to the function when it is called.

# Function Parameters/ Arguments

## Number of Arguments

- By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less.

```
def my_function(fname, lname):
  print(fname + " " + lname)


my_function("BT","CD")
```

## Arbitrary Arguments, *args

- If you do not know how many arguments that will be passed into your function, add a * before the parameter name in the function definition.
- This way the function will receive a tuple of arguments, and can access the items accordingly:
- **def my_function(*kids):**
- **print("The youngest child is " + kids[2])**

- **my_function("Emil", "Tobias", "Linus")**

# Function Parameters/ Arguments

Arbitrary Arguments, *args

Example :

```python
def add(*numbers):
    total = 0
    for num in numbers:
        total = total+ num
    return total
print(add(2, 3))
print(add(2, 3, 5))
print(add(2, 3, 5, 7))
print(add(2, 3, 5, 7, 9))
```

# Function Return Values

**Return Values**

**A Python function may or may not return a value. If we want our function to return some value to a function call, we use the return statement. For example,**

- ▪ <sub>A</sub> **return statement is used to end the execution of the function call and "returns" the result (value of the expression following the return keyword) to the caller.**
- ▪ **The statements after the return statements are not executed.**
- ▪ **To let a function return a value, use the return statement:**

```
def fun():
        statements . .

        return [expression]
```

**Return Values**

```python
def find_square(num):
    result = num * num
    return result
# function call
find_square(3)
```

# Function Parameters/ Arguments

## Arbitrary Keyword Arguments, **kwargs

▪ If you do not know how many **keyword arguments** that will be passed into your function, add two asterisk: ** before the parameter name in the function definition.
▪ This way the function will receive a *dictionary* of arguments, and can access the items accordingly:

```
def total_fruits(**kwargs):
    print(kwargs, type(kwargs))


total_fruits(banana=5, mango=7, apple=8)
```

## Default Values

- Default values indicate that the function argument will take that value if no argument value is passed during the function call.
- The default value is assigned by using the assignment(=) operator of the form *keywordname*=value.

```python
def my_function(country = "India"):
    print("I am from " + country)

my_function("Sweden")
my_function("Norway")
my_function()
my_function("Brazil")
```

# Function Parameters/ Arguments

## Global Variables

- Variables that are created outside of a function (as in all of the examples above) are known as global variables.
- Global variables can be used by everyone, both inside of functions and outside.

Example 1
```python
x = "awesome"

def myfunc():
  print("Python is " + x)

myfunc()
```

Example 2
```python
x = "awesome"

def myfunc():
  x = "fantastic"
  print("Python is " + x)

myfunc()

print("Python is " + x)
```

# Function Parameters/ Arguments

## Python Constants

- A constant is a special type of variable whose value cannot be changed.
- In Python, constants are usually declared and assigned in a [module](#) (a new file containing variables, functions, etc which is imported to the main file).
- Let's see how we declare constants in separate file and use it in the main file,

**File name : constant.py**

```
# declare constants
PI = 3.14
GRAVITY = 9.8
```

**File name main.py**

```
# import constant file we created above
import constant
print(constant.PI) # prints 3.14
print(constant.GRAVITY) # prints 9.8
```

# Files and Exceptions

Python too supports file handling and allows users to handle files i.e., to read and write files, along with many other file handling options, to operate on files.

# File Handling

The key function for working with files in Python is the `open()` function.
The `open()` function takes two parameters; *filename*, and *mode*.
There are four different methods (modes) for opening a file:

Before performing any operation on the file like reading or writing, first, we have to open that file. For this, we should use Python's inbuilt function open() but at the time of opening, we have to specify the mode, which represents the purpose of the opening file.

```
f = open(filename, mode)
```

# Files and Exceptions

### Mode is supported:

1. **r:** open an existing file for a read operation.
2. **w:** open an existing file for a write operation. If the file already contains some data then it will be overridden but if the file is not present then it creates the file as well.
3. **a:** open an existing file for append operation. It won't override existing data.
4. **r+:** To read and write data into the file. The previous data in the file will be overridden.
5. **w+:** To write and read data. It will override existing data.
6. **a+:** To append and read data from the file. It won't override existing data.

# Files and Exceptions

```
# a file named "mytext", will be opened with the reading mode.
file = open('myfile.txt', 'r')
# This will print every line one by one in the file
for each in file:
        print (each)
```

There is more than one way to read a file in Python. If you need to extract a string that contains all characters in the file then we can use **file.read()**. The full code would work like this:

```
# Python code to illustrate read() mode
file = open("file.txt", "r")
print (file.read())
```

# Files and Exceptions

Another way to read a file is to call a certain number of characters like in the following code the interpreter will read the first five characters of stored data and return it as a string:

```
# Python code to illustrate read() mode character wise
file = open("file.txt", "r")
print (file.read(5))
```

# Files and Exceptions

**Creating a file using write() mode**

Let's see how to create a file and how to write mode works, so in order to manipulate the file, write the following in your Python environment:

```python
# Python code to create a file
file = open('myfile.txt','w')
file.write("This is the write command")
file.write("It allows us to write in a particular file")
file.close()
```

**Creating a file using append() mode**
It is same as write mode but the previous content remains as it is and the new content appended at the end of the file

```
# Python code to illustrate append() mode
file = open('geek.txt', 'a')
file.write("This will add this line")
file.close()
```

**There are also various other commands in file handling that is used to handle various tasks like:**

```
rstrip(): This function strips each line of a file off spaces
from the right-hand side.
lstrip(): This function strips each line of a file off spaces
from the left-hand side.
```

# Files and Exceptions

**Using write along with the with() function**

We can also use the write function along with the  with() function:

```
# Python code to illustrate with() alongwith write()
with open("file.txt", "w") as f:
      f.write("Hello World!!!")
```

**split() using file handling**

We can also split lines using file handling in Python. This splits the variable when space is encountered.
You can also split using any characters as we wish. Here is the code:

```
# Python code to illustrate split() function
with open("file.text", "r") as file:
      data = file.readlines()
      for line in data:
            word = line.split()
            print (word)
```

# Exceptions

Error in Python can be of two types i.e. <u>Syntax errors and Exceptions</u>. Errors are the problems in a program due to which the program will stop the execution. On the other hand, exceptions are raised when some internal events occur which changes the normal flow of the program.

**Difference between Syntax Error and Exceptions**
**Syntax Error:** As the name suggests this error is caused by the wrong syntax in the code. It leads to the termination of the program.

```
# initialize the amount variable
amount = 10000

# check that You are eligible to
# purchase Dsa Self Paced or not
if(amount > 2999)
print("You are eligible to purchase Dsa Self Paced")
```

```
File "/home/ac35380186f4ca7978956ff46697139b.py", line 4
    if(amount>2999)
                  ^
SyntaxError: invalid syntax
```

# Exceptions

**Exceptions:** Exceptions are raised when the program is syntactically correct, but the code resulted in an error. This error does not stop the execution of the program, however, it changes the normal flow of the program.

```
# initialize the amount variable
marks = 10000

# perform division with 0
a = marks / 0
print(a)
```

```
Traceback (most recent call last):
  File "/home/f3ad05420ab851d4bd106ffb04229907.py", line 4, in <module>
    a=marks/0
ZeroDivisionError: division by zero
```

# Exceptions

**Try and Except Statement – Catching Exceptions**
Try and except statements are used to catch and handle exceptions in Python. Statements that can raise exceptions are kept inside the try clause and the statements that handle the exception are written inside except clause.

```python
# Python program to handle simple runtime error
#Python 3

a = [1, 2, 3]
try:
    print ("Second element = %d" %(a[1]))

    # Throws error since there are only 3 elements in array
    print ("Fourth element = %d" %(a[3]))

except:
    print ("An error occurred")
```

**Output**
Second element = 2
An error occurred

# Exceptions

## Catching specific exception in Python

```
# Program to handle multiple errors with one
# except statement
# Python 3

def fun(a):
    if a < 4:

        # throws ZeroDivisionError for a = 3
        b = a/(a-3)

    # throws NameError if a >= 4
    print("Value of b = ", b)
```

```
try:
    fun(3)
    fun(5)

# note that braces () are necessary here for
# multiple exceptions
except ZeroDivisionError:
    print("ZeroDivisionError Occurred and Handled")
except NameError:
    print("NameError Occurred and Handled")
```

The output above is so because as soon as python tries to access the value of b, NameError occurs.

# Exceptions

**Try with Else Clause**

In python, you can also use the else clause on the try-except block which must be present after all the except clauses. The code enters the else block only if the try clause does not raise an exception.

```python
# Program to depict else clause with try-except
# Python 3
# Function which returns a/b
def AbyB(a , b):
    try:
        c = ((a+b) / (a-b))
    except ZeroDivisionError:
        print ("a/b result in 0")
    else:
        print (c)


# Driver program to test above function
AbyB(2.0, 3.0)
AbyB(3.0, 3.0)
```

**Output**
```
-5.0 a/b result in 0
```

# Exceptions

**Finally Keyword in Python**

Python provides a keyword <u>finally</u>, which is always executed after the try and except blocks. The final block always executes after normal termination of try block or after try block terminates due to some exception.

# Python program to demonstrate finally

# No exception Exception raised in try block
try:
    k = 5//0 # raises divide by zero exception.
    print(k)

# handles zerodivision exception
except ZeroDivisionError:
    print("Can't divide by zero")

finally:
    # this block is always executed
    # regardless of exception generation.
    print('This is always executed')

**Output**
```
Can't divide by zero
This is always executed
```

# Exceptions

**Raising Exception**
The <u>raise statement</u> allows the programmer to force a specific exception to occur. The sole argument in raise indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from Exception).

```
# Program to depict Raising Exception

try:
    raise NameError("Hi there") # Raise Error
except NameError:
    print ("An exception")
    raise # To determine whether the exception was raised or
not
```

```
Traceback (most recent call last): File
"/home/d6ec14ca595b97bff8d8034bbf212a9f.py", line
5, in <module> raise NameError("Hi there") # Raise
Error NameError: Hi there
```