**Unit-II**

Functions: Defining a function, Calling a function, returning multiple values from a function, functions are first class objects, formal and actual arguments, positional arguments, recursive functions. Exceptions: Errors in a Python program, exceptions, exception handling, types of exceptions, the except block, the assert statement, user-defined exceptions.

# Functions

A function is a block of organized, reusable code that is used to perform a single, related action. As you already know, Python gives you many built-in functions like print(), etc. but you can also create your own functions. These functions are called *user-defined functions.*

## Defining a function:-

You can define functions to provide the required functionality. Here are simple rules to define a function in Python.

- 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
   return [expression]
```

By default, parameters have a positional behavior and you need to inform them in the same order that they were defined.

Example

The following function takes a string as input parameter and prints it on standard screen.

```
def printme( str ):
   "This prints a passed string into this function"
   print str
   return
```

## Calling a function:-

Defining a function only gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code.

Once the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt. Following is the example to call printme() function −

```
# Function definition is here
def printme( str ):
   "This prints a passed string into this function"
   print str
   return

# Now you can call printme function
printme("I'm first call to user defined function!")
printme("Again second call to the same function")
```

When the above code is executed, it produces the following result −

```
I'm first call to user defined function!
Again second call to the same function
```

## Returning multiple values from a function:-

Returning multiple values from a function is quite cumbersome in C and other languages, but it is very easy to do in Python.

You can return multiple values by simply `return` them separated by commas.

As an example, define a function that returns a string and a number as follows: Just write each value after the `return`, separated by commas.

```
def test():
   return 'abc', 100
```

In Python, comma-separated values are considered tuples without parentheses, except where required by syntax. For this reason, the function in the above example returns a tuple with each value as an element.

```
result = test()
```

```
print(result)
print(type(result))
# ('abc', 100)
# <class 'tuple'>


print(result[0])
print(type(result[0]))
# abc
# <class 'str'>

print(result[1])
print(type(result[1]))
# 100
# <class 'int'>
```

Unpack a tuple / list in Python

You can unpack and assign multiple return values to different variables

```
a, b = test()

print(a)
# abc

print(b)
# 100
```

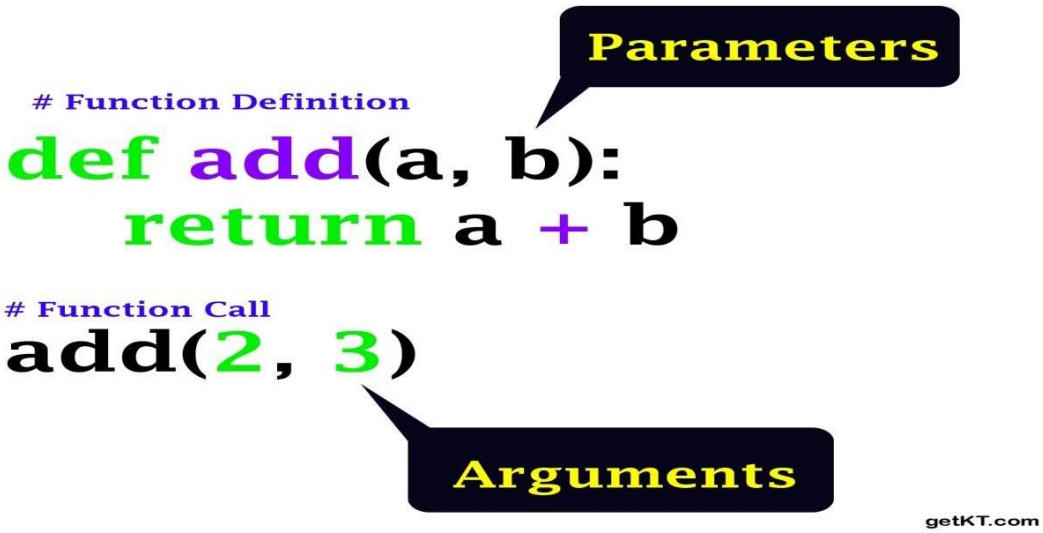
Using `[]` returns `list` instead of `tuple`.

```
def test_list():
    return ['abc', 100]


result = test_list()


print(result)
print(type(result))
# ['abc', 100]
# <class 'list'>
```

**Formal and Actual arguments:-**

**Arguments** are values that are passed into function (or method) when the calling function
**Parameters** are variables(identifiers) specified in the (header of) function definition

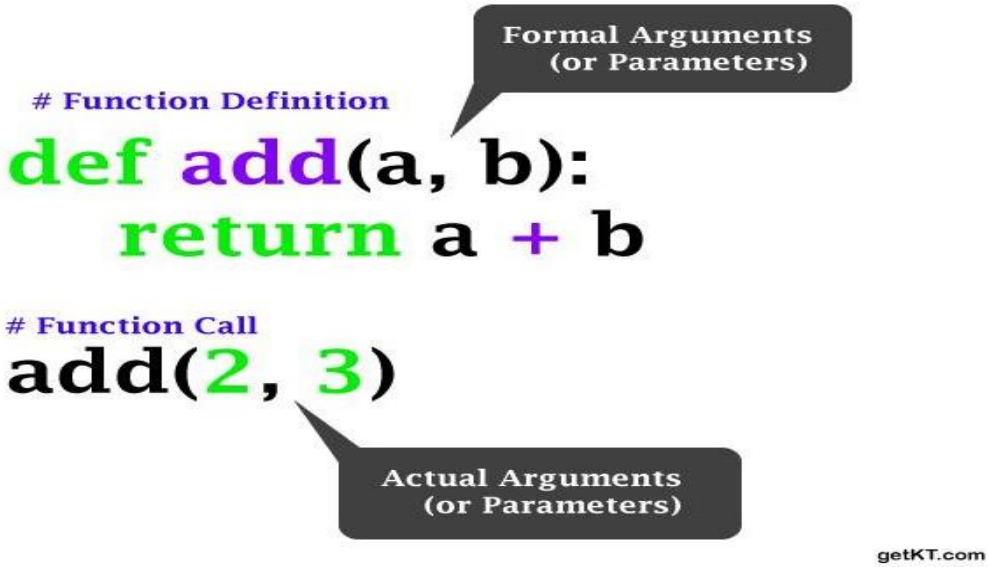
Following image shows difference between parameters and arguments.




Function Parameters VS Arguments

**Formal arguments** are identifiers used in the function definition to represent corresponding actual arguments.

**Actual arguments** are values(or variables)/expressions that are used inside the parentheses of a function call.



## Positional Arguments:-

An argument is a variable, value or object passed to a function or method as input. Positional arguments are arguments that need to be included in the proper position or order.

The first positional argument always needs to be listed first when the function is called. The second positional argument needs to be listed second and the third positional argument listed third, etc.

An example of positional arguments can be seen in Python's complex() function. This function returns a complex number with a real term and an imaginary term. The order that numbers are passed to the complex() function determines which number is the real term and which number is the imaginary term.

If the complex number 3 + 5j is created, the two positional arguments are the numbers 3 and 5. As positional arguments, 3 must be listed first, and 5 must be listed second.

In [1]:

complex(3, 5)

Out[1]:

(3+5j)

On the other hand, if the complex number 5 + 3j needs to be created, the 5 needs to be listed first and the 3 listed second. Writing the same arguments in a different order produces a different result.

In [2]:

complex(5, 3)

Out[2]:

(5+3j)

Positional Arguments Specified by an Iterable -

Positional arguments can also be passed to functions using an iterable object. Examples of iterable objects in Python include lists and tuples. The general syntax to use is:

function(*iterable)

Where function is the name of the function and iterable is the name of the iterable preceded by the asterisk * character.

An example of using a list to pass positional arguments to the complex() function is below. Note the asterisk * character is included before the term_list argument.

In [3]:

```
term_list = [3, 5]
complex(*term_list)
```

Out[3]:

(3+5j)

**Functions are first class objects:-**

First Class objects are those objects, which can be handled uniformly.

First Class objects can be stored as Data Structures, as some parameters of some other functions, as control structures etc.

The first class objects are program entity which have these five characteristics:

1. Can be created at runtime.
2. Can be assigned to a variable.
3. Can be passed as a argument to a function.
4. Can be return as a result from a function.
5. Can have properties and methods

Let's see each point one by one through code.

1. Can be created at runtime

```
add = eval("lambda a,b : a + b")
print(add(10,20)) # logs: 30
print(type(add)) # logs: <class 'function'>
```

Here add function is created at runtime and you can also see it is instance of a function class.

2. Can be assigned to a variable

```
getSquare = lambda value: value * value
[print(getSquare(value), end=' ') for value in [10,15,20]]
# logs: 100 225 400
```

Here anonymous function, lambda value: value * value is assigned to variable getSqaure, now we can use getSqaure variable as a function in our program.

3. Can be passed as a parameter to a function

```
import math
def customFilter(arr,pred):
    return [value for value in arr if pred(value)]
arr = [100,23,45,75,225,36]
isPerfectSqaure = lambda x: int(math.sqrt(x)) ** 2 == x
print(customFilter(arr,isPerfectSqaure)) #logs: [100, 225, 36]
print(customFilter(arr,lambda x: x > 50)) #logs: [100, 75, 225]
```

We have created function customFilter which take arr(type list) and pred(type function) as arguments. In customFilter we can pass any function which take one argument and return bool value as a result.

For example we are passing isPerfectSquare function to find elements which are perfect square and in second case passing anonymous function to find elements which are greater than 50.

4. Can be return as a result of a function

```
from time import time
def wrapper(fun):
    def inner(*args):
        start = time()
        result = fun(*args)
        end = time()
        print(f'Total time taken: {end - start} s')
        return result
    return inner
getSum = wrapper(sum)
print(type(getSum))
# logs: <class 'function'>
print(getSum(list(range(1,2000000))))
# logs
# Total time taken: 0.1580057144165039 s
```

```
# 1999999000000
```

Here we have created wrapper function, which take function as a argument and return function as a result. What this wrapper do is that it logs time taken to execute the function.

Printing type of getSum shows it is instance of a function class.

5. Can have properties and methods

```
def getCube(num):
   return num * num * num
# Printing properties and methods getCube function object have
print(dir(getCube))
#logs: ['__annotations__', '__call__', '__class__',...] Truncated #due to space
# Printing name property of getCube function object
print(getCube.__name__) #logs: getCube
# Calling getCube function using __call___method
print(getCube.__call__(10)) #logs: 1000
```

Functions in Python are First Class Objects because they satisfies these five characteristics.

## **Recursive functions:-**

It is legal for one function to call another; it is also legal for a function to call itself. It may not be obvious why that is a good thing, but it turns out to be one of the most magical things a program can do. For example, look at the following function:

```
def countdown(n):
        if n <= 0:
                print('Blastoff!')
        else:
                print(n)
                countdown(n-1)
```

If n is 0 or negative, it outputs the word, “Blastoff!” Otherwise, it outputs n and then calls a function named countdown—itself—passing n-1 as an argument.

What happens if we call this function like this?

```
>>> countdown(3)
```

The execution of countdown begins with n=3, and since n is greater than 0, it outputs the value 3, and then calls itself...

The execution of countdown begins with n=2, and since n is greater than 0, it outputs the value 2, and then calls itself...

The execution of countdown begins with n=1, and since n is greater than 0, it outputs the value 1, and then calls itself...

The execution of countdown begins with n=0, and since n is not greater than 0, it outputs the word, “Blastoff!” and then returns.

The countdown that got n=1 returns.

The countdown that got n=2 returns.

The countdown that got n=3 returns.

And then you’re back in __main__. So, the total output looks like this:

3

2

1

Blastoff!

A function that calls itself is recursive; the process of executing it is called recursion.

Stack Diagrams for Recursive Functions -

A stack diagram is used to represent the state of a program during a function call. The same kind of diagram can help interpret a recursive function.

Every time a function gets called, Python creates a frame to contain the function’s local variables and parameters. For a recursive function, there might be more than one frame on the stack at the same time.

Figure 5-1 shows a stack diagram for countdown called with n = 3.

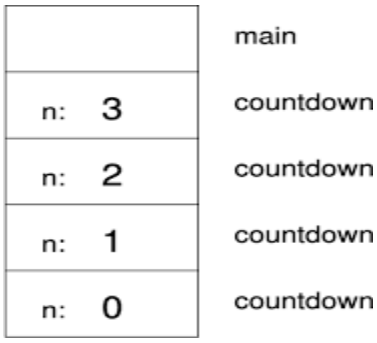



Figure 5-1. Stack diagram.

As usual, the top of the stack is the frame for __main_. It is empty because we did not create any variables in __main___or pass any arguments to it.

The four countdown frames have different values for the parameter n. The bottom of the stack, where n=0, is called the base case. It does not make a recursive call, so there

are no more frames.

As an exercise, draw a stack diagram for print_n called with s = 'Hello' and n=2.

Then write a function called do_n that takes a function object and a number, n, as

arguments, and that calls the given function n times.

## Exceptions

**Errors in a Python program:-**

When you are debugging, you should distinguish among different kinds of errors in order to track them down more quickly:

• Syntax errors are discovered by the interpreter when it is translating the source code into byte code. They indicate that there is something wrong with the structure of the program. Example: Omitting the colon at the end of a def statement generates the somewhat redundant message SyntaxError: invalid syntax.

• Runtime errors are produced by the interpreter if something goes wrong while the program is running. Most runtime error messages include information about where the error occurred and what functions were executing. Example: An infinite recursion eventually causes the runtime error maximum recursion depth exceeded.

• Semantic errors are problems with a program that runs without producing error messages but doesn't do the right thing. Example: An expression may not be evaluated in the order you expect, yielding an incorrect result.

**Syntax errors -**

Syntax errors are the most basic type of error. They arise when the Python parser is unable to understand a line of code. Syntax errors are almost always fatal, i.e. there is almost never a way to successfully execute a piece of code containing syntax errors.

In IDLE, it will highlight where the syntax error is. Most syntax errors are typos, incorrect indentation, or incorrect arguments. If you get this error, try looking at your code for any of these.

Example:

print "Hello World!"

In this first example, we forget to use the parenthesis that are required by print(). Python does not understand what you are trying to do.

Here are some ways to avoid the most common syntax errors:

1. Make sure you are not using a Python keyword for a variable name.

2. Check that you have a colon at the end of the header of every compound statement, including for, while, if, and def statements.

3. Make sure that any strings in the code have matching quotation marks. Make sure that all quotation marks are straight quotes, not curly quotes.

4. If you have multiline strings with triple quotes (single or double), make sure you have terminated the string properly. An unterminated string may cause an invalid token error at the end of your program, or it may treat the following part of the program as a string until it comes to the next string. In the second case, it might not produce an error message at all!

5. An unclosed opening operator—(, {, or [—makes Python continue with the next line as part of the current statement. Generally, an error occurs almost immediately in the next line.

6. Check for the classic = instead of == inside a conditional.

7. Check the indentation to make sure it lines up the way it is supposed to. Python can handle space and tabs, but if you mix them it can cause problems. The best way to avoid this problem is to use a text editor that knows about Python and generates consistent indentation.

8. If you have non-ASCII characters in the code (including strings and comments), that might cause a problem, although Python 3 usually handles non-ASCII characters. Be careful if you paste in text from a web page or other source

**Run time errors -**

Run time errors arise when the python knows what to do with a piece of code but is unable to perform the action. Since Python is an interpreted language, these errors will not occur until the flow of control in your program reaches the line with the problem. Common example of runtime errors are using an undefined variable or mistyped the variable name.

```
day = "Sunday"
print(Day)
```

Output

```
Traceback (most recent call last):
  File "C:/Users/91981/AppData/Local/Programs/Python/Python38/hello.py", line 2, in
    print(Day)
NameError: name 'Day' is not defined
```

**Semantic errors (or) Logical errors -**

These are the most difficult type of error to find, because they will give unpredictable results and may crash your program. A lot of different things can happen if you have a logic error.

Example: For example, perhaps you want a program to calculate the average of two numbers and get the result like this :

```
x = 3
y = 4
average = x + y / 2
print(average)
```

Output

5.0

## Exceptions:-

An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions. In general, when a Python script encounters a situation that it cannot cope with, it raises an exception.

An exception is **a Python object that represents an error**.

Even if the syntax of a statement or expression is correct, it may still cause an error when executed. Python exceptions are errors that are detected during execution and are not unconditionally fatal

## Exception handling:-

The programs usually do not handle exceptions, and result in error messages as shown here:

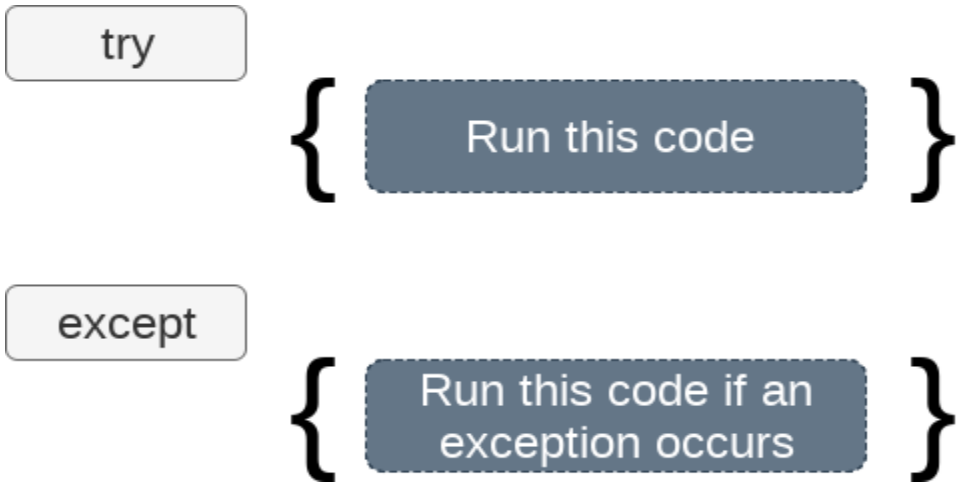
```
a = 2
b = 'DataCamp'
a + b
```

output -

```
TypeError                         Traceback (most recent call last)
<ipython-input-7-86a706a0ffdf> in <module>
    1 a = 2
    2 b = 'DataCamp'
----> 3 a + b
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

### The try-expect statement

If the Python program contains suspicious code that may throw the exception, we must place that code in the **try** block. The **try** block must be followed with the **except** statement, which contains a block of code that will be executed if there is some exception in the try block.




Example 1

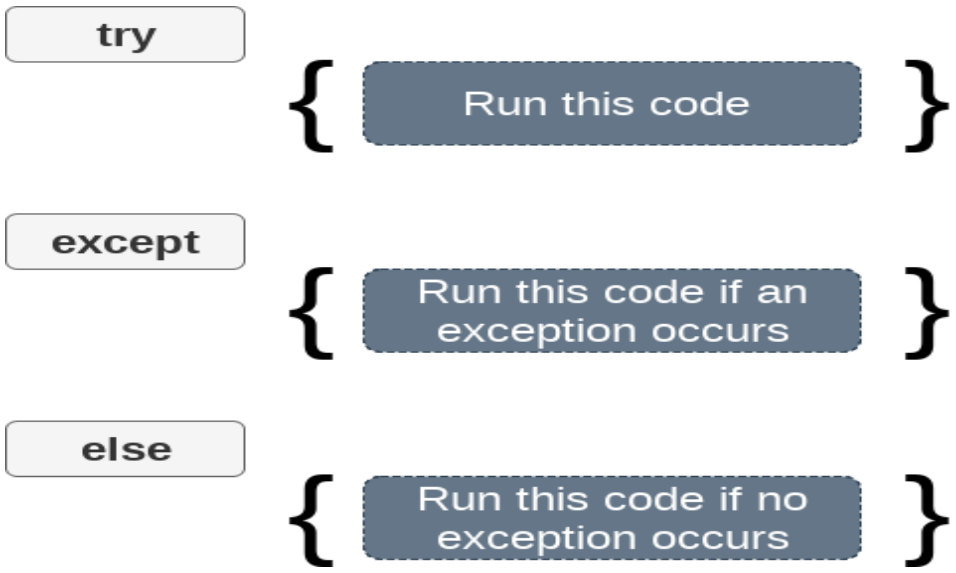
```
try:
   a = int(input("Enter a:"))
   b = int(input("Enter b:"))
   c = a/b
except:
   print("Can't divide with zero")
```

Output:

```
Enter a:10
Enter b:0
Can't divide with zero
```

### The try-expect-else statement

The syntax to use the else statement with the try-except statement is given below.




**Example 2**

```
1. try:
2.     a = int(input("Enter a:"))
3.     b = int(input("Enter b:"))
4.     c = a/b
5.     print("a/b = %d"%c)
6. except Exception:
7.     print("can't divide by zero")
8.     print(Exception)
9. else:
10.    print("Hi I am else block")
```

Output1:

```
Enter a:10
Enter b:0
can't divide by zero
<class 'Exception'>
```

Output2:

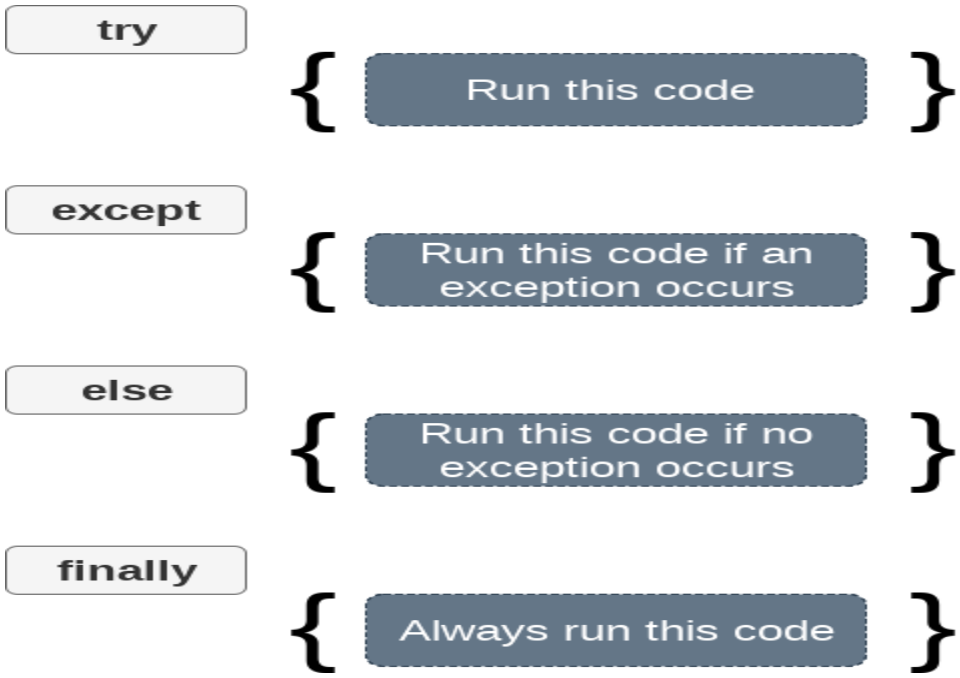
```
Enter a:12
Enter b:3
a/b = 4
Hi I am else block
```

### The try...finally block

Python provides the optional **finally** statement, which is used with the **try** statement. It is executed no matter what exception occurs and used to release the external resource. The finally block provides a guarantee of the execution.

We can use the finally block with the try block in which we can pace the necessary code, which must be executed before the try statement throws an exception.

The syntax to use the finally block is given below.




```
def divide(x, y):
        try:
                # Floor Division : Gives only Fractional
                # Part as Answer
                result = x // y
        except ZeroDivisionError:
                print("Sorry ! You are dividing by zero ")
        else:
                print("Yeah ! Your answer is :", result)
        finally:
                # this block is always executed
                # regardless of exception generation.
                print('This is always executed')

# Look at parameters and note the working of Program
```

```
divide(3, 2)
divide(3, 0)
```

Output:

```
Yeah ! Your answer is : 1
This is always executed
Sorry ! You are dividing by zero
This is always executed
```

**Raise -**

Raise exceptions in several ways by using the raise statement.

Syntax

```
raise (Exception(args(traceback)))
```

Program

```
a = 1000
if 100 < a:
  raise Exception("Sorry, the numbers above 100")
```

Output

```
Traceback (most recent call last):
  File "main.py", line 3, in <module>
    raise Exception("Sorry, the numbers above 100")
Exception: Sorry, the numbers above 100
```

## Types of exceptions:-

I. Built-in Exceptions
II. User-defined Exceptions

**Built-in Exceptions -**

There are several built-in exceptions in Python that are raised when errors occur.

In Python, all exceptions must be instances of a class that derives from BaseException.

BaseException

```
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
    +-- StopIteration
    +-- StopAsyncIteration
    +-- ArithmeticError
    |    +-- FloatingPointError
    |    +-- OverflowError
    |    +-- ZeroDivisionError
    +-- AssertionError
    +-- AttributeError
    +-- BufferError
    +-- EOFError
    +-- ImportError
    |    +-- ModuleNotFoundError
    +-- LookupError
    |    +-- IndexError
    |    +-- KeyError
    +-- MemoryError
    +-- NameError
    |    +-- UnboundLocalError
    +-- OSError
    |    +-- BlockingIOError
    |    +-- ChildProcessError
    |    +-- ConnectionError
    |    |    +-- BrokenPipeError
    |    |    +-- ConnectionAbortedError
    |    |    +-- ConnectionRefusedError
    |    |    +-- ConnectionResetError
    |    +-- FileExistsError
    |    +-- FileNotFoundError
    |    +-- InterruptedError
    |    +-- IsADirectoryError
    |    +-- NotADirectoryError
    |    +-- PermissionError
    |    +-- ProcessLookupError
    |    +-- TimeoutError
    +-- ReferenceError
    +-- RuntimeError
    |    +-- NotImplementedError
    |    +-- RecursionError
    +-- SyntaxError
    |    +-- IndentationError
```

```
|       +-- TabError
+-- SystemError
+-- TypeError
+-- ValueError
|    +-- UnicodeError
|       +-- UnicodeDecodeError
|       +-- UnicodeEncodeError
|       +-- UnicodeTranslateError
+-- Warning
    +-- DeprecationWarning
    +-- PendingDeprecationWarning
    +-- RuntimeWarning
    +-- SyntaxWarning
    +-- UserWarning
    +-- FutureWarning
    +-- ImportWarning
    +-- UnicodeWarning
    +-- BytesWarning
    +-- ResourceWarning
```

exception Exception
All built-in, non-system-exiting exceptions are derived from this class. All user-defined exceptions should also be derived from this class.

exception ArithmeticError
The base class for those built-in exceptions that are raised for various arithmetic errors: OverflowError, ZeroDivisionError, FloatingPointError.

exception AssertionError
Raised when an assert statement fails.

exception AttributeError
Raised when an attribute reference (see Attribute references) or assignment fails. (When an object does not support attribute references or attribute assignments at all, TypeError is raised.)

exception EOFError
Raised when the input() function hits an end-of-file condition (EOF) without reading any data. (N.B.: the io.IOBase.read() and io.IOBase.readline() methods return an empty string when they hit EOF.)

Exception ImportError
Raised when the import statement has troubles trying to load a module. Also raised when the “from list” in from ... import has a name that cannot be found.

exception ModuleNotFoundError
A subclass of ImportError which is raised by import when a module could not be located. It is also raised when None is found in sys.modules.

exception IndexError
Raised when a sequence subscript is out of range. (Slice indices are silently truncated to fall in the allowed range; if an index is not an integer, TypeError is raised.)

exception KeyError
Raised when a mapping (dictionary) key is not found in the set of existing keys.

exception NameError
Raised when a local or global name is not found. This applies only to unqualified names. The associated value is an error message that includes the name that could not be found.

exception TypeError
Raised when an operation or function is applied to an object of inappropriate type. The associated value is a string giving details about the type mismatch.

**User-defined Exceptions -**

Python has numerous built-in exceptions that force your program to output an error when something in the program goes wrong.

However, sometimes you may need to create your own custom exceptions that serve your purpose.

Example

```
class SalaryNotInRangeError(Exception):
   """Exception raised for errors in the input salary.

   Attributes:
      salary -- input salary which caused the error
      message -- explanation of the error
   """

   def __init_(self, salary, message="Salary is not in (5000, 15000) range"):
      self.salary = salary
      self.message = message
      super()._init_(self.message)
```

```
salary = int(input("Enter salary amount: "))
if not 5000 < salary < 15000:
   raise SalaryNotInRangeError(salary)
```

output

```
Enter salary amount: 2000
Traceback (most recent call last):
  File "main.py", line 17, in <module>
    raise SalaryNotInRangeError(salary)
__main__.SalaryNotInRangeError: Salary is not in (5000, 15000) range
```

**The assert statement**:-

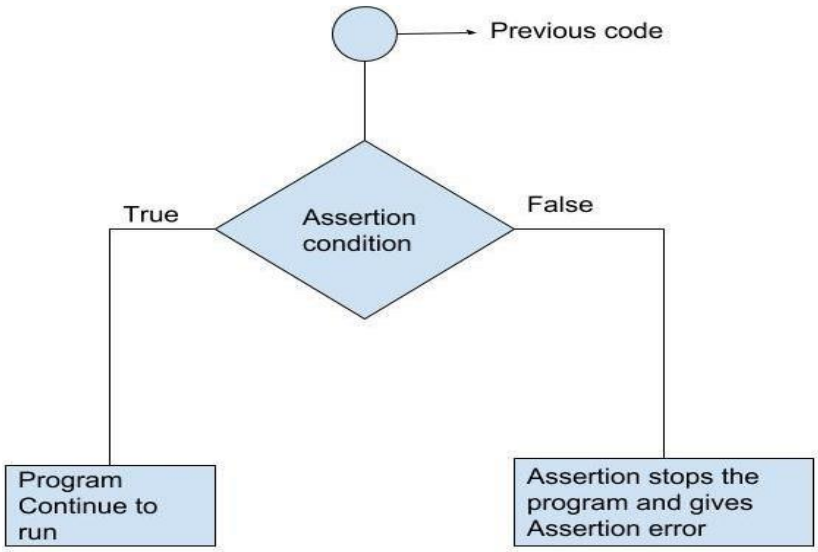
Assertions are statements that assert or state a fact confidently in your program. For example, while writing a division function, you're confident the divisor shouldn't be zero, you assert divisor is not equal to zero.

Assertions are simply boolean expressions that check if the conditions return true or not. If it is true, the program does nothing and moves to the next line of code. However, if it's false, the program stops and throws an error.

It is also a debugging tool as it halts the program as soon as an error occurs and displays it.

We can be clear by looking at the flowchart below:

Python Assert Flowchart



Python assert Statement

Python has built-in assert statement to use assertion condition in the program. assert statement has a condition or expression which is supposed to be always true. If the condition is false assert halts the program and gives an AssertionError.

Syntax for using Assert in Pyhton:

assert <condition>

assert <condition>,<error message>

In Python we can use assert statement in two ways as mentioned above.

assert statement has a condition and if the condition is not satisfied the program will stop and give AssertionError.

assert statement can also have a condition and a optional error message. If the condition is not satisfied assert stops the program and gives AssertionError along with the error message.

Example1

```
x = "hello"

#if condition returns False, AssertionError is raised:
assert x == "goodbye", "x should be 'hello'"
```

output

```
Traceback (most recent call last):
  File "demo_ref_keyword_assert2.py", line 4, in <module>
    assert x == "goodbye", "x should be 'hello'"
AssertionError: x should be 'hello'
```

Example2

```
def avg(marks):
    assert len(marks) != 0,"List is empty."
    return sum(marks)/len(marks)

mark2 = [55,88,78,90,79]
print("Average of mark2:",avg(mark2))

mark1 = []
```

```
print("Average of mark1:",avg(mark1))
```

output

```
Average of mark2: 78.0
AssertionError: List is empty.
```