Python Programming

Computer Language

- definitions
    - set of instructions (algorithm)
    - implementation of algorithm
    - helps us to interact with hardware
    - medium of communication with hardware
- types
    - based on the level
        - **low level**
            - binary (0s and 1s)
        - **middle level**
            - interacts with CPU
            - Assembly language
            - opcodes: operation code => binary
            - e.g. ADD A, B
        - **high level**
            - developer can write human understable code
            - compiler or interprter convers the human understandable to machine (CPU) understandable (ASM)
            - e.g. C++, Java, Python
    - based on how the application gets generated
        - **compiled language**
            - compile: converting human understable to machine (CPU) understandale
            - compiler: program which does compilation
            - executable:
                - program which contains only ASM instructions (machine understandable)
                - native applications
                - always platform (OS) dependent
                - faster than interpreted program
            - requires compiler
            - the entire program gets converted into executable
            - if program contains error, compiler detects these error at compilation time
            - e.g. C, C++
        - **interperted language**
            - interpretation: which converts the human understandable to machine (CPU) understandable line by line
            - interpreter: program which does interpretation

        - no executable gets generated
        - if there is any error, it will get detected at the run time
        - program will be always platform (OS) independent
        - programs will be always slower than native applications
        - e.g. html/CSS, JS, bash scripting

    - **mixed language**

        - shows behavior from both (compiled as well as interptreted)
        - uses compiler as well as interpreter
        - e.g. Java, **Python**

Introduction to Python

- is high-level language which shows behavior from both compiled as well as interpreted languages
- developed by Guido Rossum
- can be used for
    - console application
    - web application
    - ML application
    - GUI application
- the python by default imports the basic packages for using the built-in function
- python application does NOT require any entry point function
    - python is one of the scripting languages
    - the code starts execution from from top (line 1) to bottom
- python is a
    - scripting language
    - oop language
    - functional programming language
    - aspect oriented programming language

environment configuration

- versions
    - 1.x: deprecated
    - 2.x: about to be deprecated
    - 3.x: latest version

installation

- to install python on ubuntu

```
sudo apt-get install python3 python3-pip
```

- to install python on centos

```
sudo yum install python3 python3-pip
```

IDE

- **PyCharm**
    - Community Edition
    - https://www.jetbrains.com/pycharm/download/
- Spyder
- Visual Studio Code
- vim

configuration

fundamentals

identifier

- valid word which is used to perform an action
- most of the times the identifiers are lower cased
- can be
    - variable name
    - function name
    - constant
    - class name
    - keyword
- rules
    - can not start with number
        - e.g.
            - 1name is invalid identifier
            - one_name is valid identifier
    - can not contain special character like space
        - e.g.
            - first name is invalid identifier
            - first_name is valid identifier
    - may use only underscore (_)
- conventions
    - for variables: lower case
        - e.g. name, address, first_name
    - for functions: lower case with underscore
        - e.g. is_eligible_for_voting
    - for class: lower case with first letter uppercase
        - e.g. Person, Mobile

variable

- identifier used to store a value
- variable can not be declared explicitly
- syntax
    - <variable name> = <initial value>
- e.g.
    - num = 100

keyword

- reserved identifier by python
- can not use keyword for declaring variables or functions
- e.g. if, elif, else, for, while, switch
- **pass**
    - do not do anything
    - pass the control to the next line
    - used to create empty function/class
- **def**
    - used to define a function
- **return**
    - used to return a value

statement

- the one which executes
- unit of execution
- semicolon (;) is used to terminate a statement
- one statement per line does not require semicolon (;)
- BUT MULTIPLE STATEMENTS ON ONE LINE MUST USE SEMICOLON (;)
- types
    - assignment statement
    - declaration statement
    - function call
    - control statement
    - conditional statement
    - comment
        - ignored while execution
        - to create comment use symbol #
- **RULE**
    - if the code has any synctical error, the python compiler will not generate the byte codes [the syntactical errors will be detected at the time compilation]

```
print("hello 1")
```

```
print("hello 2"

# this code will generate SyntaxError
# even the first line will NOT get executed
```

- if the code has any run time error, the compilation will not detect any error and program will execute till the line where the error is detected

```
print("hello 1")
printf("hello 2")

# this code will generate NameError
# the first line will get executed and code will stoop on the line 2
```

block

- group of statements
- use space(s)/tab(s) [indentation] to create a block
- e.g. function, if, else, while

control statements

- **if..else**
  - used to check a condition
  - e.g.

```
if p1 % 2 == 0:
    print(f"{p1} is even")
else:
    print(f"{p1} is not even")
```

data types

- in python, all data types are **inferred**
- in python, all data types are assigned implicitly
- data types will get assigned automatically (by python itself) by looking at the CURRENT value of the variable
- you can not declare a variable with explicit data type
- e.g.

```
# can not declare explicit
# int num = 100
```

- types
  - **int**
    - represents the whole numbers (+ve or -ve)
    - e.g.
      - num = 100
      - myvar = -10
  - **float**
    - represents a value with decimal
    - e.g.
      - salary = 10.60
  - **str**
    - represents a string
    - to create a string value use
      - single quotes
        - used to create single line string
        - e.g.

```
name = 'steve'
```

      - double quotes
        - used to create single line string
        - e.g.

```
last_name = "jobs"
```

      - tripe double quotes
        - used to create multi-line string
        - e.g.

```
address = """
    House no 100,
    XYZ,
    pune 411056,
    MH, India.
"""
```

  - **bool**
    - represents boolean value
    - can contain one of the two values [True/False]

        - e.g.

```
can_vote = True
```

    - **complex**
    - **object**

operators

mathematical

- + : addition/string concatination
- - : subtraction
- * : multiplication
- / : true division (float)
- //: floor division (int)
- **: power of

comparison

- == : equal to
- != : not equal
- > : greater than
- < : less than
- >=: greater than or equal to
- <=: less than or equal to

logical

- and:
    - logical and operator
    - returns true only when both the conditions are true
    - rule
        - true and true => true
        - true and false => false
        - false and true => false
        - false and false => false

```
if (age > 20) and (age < 60):
    print(f"{age} is within the limit")
else:
    print(f"{age} is not within the limit")
```

- or:
    - logical or operator
    - returns true when one of the conditions is true
    - rule
        - true or true => true
        - true or false => true
        - false or true => true
        - false or false => false

```
if (age > 20) or (age < 60):
    print(f"{age} is within the limit")
else:
    print(f"{age} is not within the limit")
```

function

- named block
- can be used to reuse the code
- in python, function name is treated as a variable (the type of such variable is function)
- function uses c calling conventions

```
def function_1():
  pass

# type of function_1: function
print(f"type of function_1: {type(function_1)}")
```

scope

- **global**
    - declared outside of any function
    - such variables can be accessed anywehere (outside or inide of any function) in the code (in the same file)
    - by default global variables are not modifyable inside function(s)
    - use **global** keyword to make the global variables modifiyable
    - e.g.

```
num = 100
```

```
# gobal variable
# num = 100
print(f"outside function num = {num}")

def function_1():
  # num = 100
  print(f"inside function_1, num = {num}")

def function_2():
  # the num will refer to the global copy
  global num

  # modify the gobal variable
  num = 200
```

- **local**
  - variable declared inside a function
  - the variable will be accessible only within the function (in which, it is declared)
  - local variables will NOT be accessible outside the function
  - e.g.

```
def function_1():
  num = 100
  print(f"inside function_1, num = {num}")

function_1()

# the statement will generate NameError as
# num is a local variable
# print(f"outside function_1, num = {num}")
```

custom

- also known as user defined function
- e.g.

```
# function declaration
def function_1():
    print("inside function_1")

# function invocation
```

```
function_1()
```

function parameters

- **positional parameters**
  - do not have parameter name whlie making the function name
  - the values will get assigned to the parameters from left to right
  - the position of paramter is very important
  - e.g.

```
def function_3(num1, num2, num3):
    print(f"num1 = {num1}, num2 = {num2}

# num1 = 10, num2 = 20, num3 = 30
function_3(10, 20, 30)
```

- **named parameters**
  - the function call will contain the parameter name along with the parameter value
  - position of the named parameter is not important
  - e.g.

```
def function_3(num1, num2, num3):
    print(f"num1 = {num1}, num2 = {num2}

# num1 = 10, num2 = 20, num3 = 30
function_3(num1=10, num2=20, num3=30)
function_3(num2=20, num3=30, num1=10)
function_3(num3=30, num2=20, num1=10)
```

- **optional parameters**
  - a function can set a default value for a parameter
  - caller does not need to pass a value for such parameters
  - the parameter having default value becomes optional (caller may or may not pass the value for such parameter)

```
# p2 has a default value = 50
def function_1(p1, p2=50):
```

```
    print(f"{p1}, {2}")

# p1 = 10, p2 = 50
function_1(10)

# p1 = 10, p2 = 20
function_1(10, 20)
```

function types

- **empty function**
  - function without a body

```
def empty_function():
    pass
```

- **parameterless function**
  - function which does not accept any parameter

```
def function_1():
    print("inside function_1")
```

- **parameterized function**
  - function which accepts at least one parameter
  - e.g.

```
def function_1(p1):
    print(f"p1 = {p1}, type = {type(p1)}")

function_1(10)  # p1 = int
function_1("10")  # p1 = str
function_1(True)  # p1 = bool
```

- **non-returning function**
  - function which does not return any function
  - e.g.

```


def function_1():
```

```
    print("inside function_1")
```

- a non-returning function always will return None
- e.g.

```
def function_1():
    print("inside function_1")

result = function_1()

# result = None
print(f"result = {result}")
```

- **returning function**
  - function which returns a value
  - e.g.

```
# returning function
def add(p1, p2):
    print("inside add")
    return p1 + p2

# capture result
addition = add(30, 50)  # 80
```

- **nested function**
  - function within a function
  - is also known as inner or local function
  - the inner function can be called only within the function in which it is declared
  - e.g.

```
def outer():
  def inner():
    pass

  # inner is callable only within outer
  inner()

outer()

# can not access inner outside the outer
```

```
# inner()
```

- properties
  - inner function can access all the members of outer function

```
def outer():
  num = 100
  def inner():
    # num = 100
    print(f"num = {num}")

  inner()

outer()
```

  - outer function can not access any members of inner function

```
def outer():
  def inner():
    num = 100
    print(f"num = {num}")

  inner()

  # can not access the inner functions' local variable
  # print(f"num = {num}")

outer()
```

  - outer function can have as many inner function as required
  - outer function can have a hierarchy of inner functions

- **variable length argument function**
  - function which accepts any number of arguments
  - when the function gets called
    - the positional parameters get collected in a tuple (args)
    - the named parameters get collected in a dictionary (kwargs)
  - e.g.

```
```

```
def va_function(*args, **kwargs):
  print(f"args = {args}, type = {type(args)}")
  print(f"kwargs = {kwargs}, type = {type(kwargs)}")

# args = (10, 20)
va_function(10, 20)

# kwargs = {'p1': 10, 'p2': 20}
va_function(p1=10, p2=20)

# args = (10, 20)
# kwargs = {'p1': 30, 'p2': 40}
va_function(10, 20, p1=30, p2=40)
```

- **function alias**
  - another name given to an existing function
  - similar to function pointer in C
  - e.g.

```
def function_2():
    print("inside function_2")

# function alias
my_function = function_2


# inside function_2
function_2()

# inside function_2
my_function()
```

lambda functions

- used to create anonymous function
- syntax:
  - lambda <param>: <body>
- rules
  - lambda **must** accept at least one parameter
  - lamdba **must** have only one statement in the body
  - the body statement **must** return a value
- e.g.

```
square = lambda x: x ** 2

# square of 10 = 100
print(f"square of 10 = {square(10)}")
```

swap two variables

- in python, the premitive data types (int, float, bool, string, complex), will be passed as values
- e.g.

```
n1, n2 = 100, 200

# n1 = 200, n2 = 100
n1, n2 = n2, n1
```

functional programming language

- language in which, the functions are treated as variables
- language in which, the function are first-class citizens
    - functions are treated as variables

```
def function_2():
    print("inside function_2")

# function alias
my_function = function_2

def add(p1, p2):
  print(f"addition = {p1 + p2}")

def multiply(p1, p2):
  print(f"multiplication = {p1 * p2}")

# collection of functions
functions = (add, multiply)

for function in functions:
  # both add and multiple will get called
  function(20, 30)
```

- function can be passed as a parameter to another

```
def add(p1, p2):
  print(f"addition = {p1 + p2}")

def executor(function):
  function(10, 20)


# executor will receive address of add
# and will call add with 10, 20 parameters
executor(add)
```

- function can be returned as a return value from another function

```
def add(p1, p2):
  print(f"addition = {p1 + p2}")

def my_function():
  # my_function is returning add as a return value
  return add

result = my_function()

# addition = 50
result(10, 40)
```

map

- used to process [new value will be generated based on existing one] all the values of a collection
- map always returns a new value
- map has to accept a parameter (which will be every value from the collection)
- e.g.

```
def square(num):
  return num ** 2

numbers = [1, 2, 3, 4, 5]

# [1, 4, 9, 16, 25]
squares = list(map(square, numbers))
```

filter

- used to filter [will remove values which do not satisfy the condition] the values from a collection
- filter always returns original values (when the function returns True)
- e.g.

```
def is_even(num):
  return num % 2 == 0

numbers = [1, 2, 3, 4, 5]

# [2, 4]
squares = list(filter(is_even, numbers))
```

list comprehension

- syntax in python, to generate a new list
- similar to map and filter
- syntax:
  - [<out> <for..in loop>]
  - [<out> <for..in loop> <criteria>]
- e.g.

```
# list comprehension similar to map

numbers = [1, 2, 3, 4, 5]

# [1, 2, 3, 4, 5]
numbers_2 = [ number for number in numbers ]

# [2, 4, 8, 8, 10]
numbers_double = [number * 2 for number in numbers]

# [1, 4, 9, 16, 25]
squares = [number ** 2 for number in numbers]
```

```
# list comprehension similar to filter

numbers = [1, 2, 3, 4, 5]
```

```
  # [2, 4]
  even = [number for number in numbers if number % 2 == 0]

  # [1, 3, 5]
  odd = [number for number in numbers if number % 2 != 0]
```

```
  # list comprehension similar to filter + map

  numbers = [1, 2, 3, 4, 5]

  # [4, 16]
  square_even_numbers = [number ** 2 for number in numbers if number % 2
== 0]

  # [1, 27, 125]
  cube_odd_numbers = [number ** 3 for number in numbers if number % 2 !=
0]
```

string

- collection of characters
- any data type can be converted to string by calling str() function
- operations
  - **capitalize()**
    - make the first letter capital
  - **casefold()**
    - make all the characters lower case
  - **center()**
    - makes the string appear in the center of number of characters
    - e.g.

```
#    report
print("report".center(11))
```

  - **count()**

        - returns the number of occurrences of a substring
    - **endswith()**
        - returns if string ends with a value
    - **expandtabs()**
    - **find()**
    - **format()**
        - used to format a string
        - format replaces the value of variables with the positions of {}
        - e.g.

```
num_1 = 10.56346567
num_2 = 1000000

# formatted string
print(f"num_1 = {num_1}, num_2 = {num_2}")

# non-formatted string
print("num_1 = {}, num_2 = {}".format(num_1, num_2))
print("num_1 = {0}, num_2 = {1}".format(num_1, num_2))
print("num_2 = {1}, num_1 = {0}".format(num_1, num_2))
```

        - format types
            - :<
                - left aligned string
                - e.g. print("name: {0:<10}".format(name))
            - :>
                - right aligned string
                - e.g. print("name: {0:>10}".format(name))
            - :^
                - center aligned string
                - e.g. print("name: {0:^10}".format(name))
            - :=
            - :+
                - show positive/negative symbol
            - :-
                - show minus only for negative values
            - :<space>

        - :,
            - show 1000 separater (,)
        - :_
            - show 1000 separater (_)
        - :b
            - convert that value binary number system
        - :d
            - convert the value in decimal number system
        - :e
            - scientific format (lower case e)
        - :E
            - scientific format (upper case e)
        - :f
            - formatting float
            - e.g. print("num_1 = {:0.2f}".format(num_1))
        - 😮
            - convert the value in octal number system
        - :x
            - convert the value in hex number system (lower case letters)
        - :X
            - convert the value in hex number system (upper case letters)
        - :n
            - convert the value in decimal
        - :%
            - used to convert a number to percentage
- **index()**
    - returns the position of a subtring
- **isalnum()**
- **isapha()**
- **isdecimal()**
- **isdigit()**
- **islower()**
- **isspace()**
- **isupper()**
- **join()**
    - joins a collection to create a string
- **lower()**
    - make all the characters lower case

    - **replace()**
    - **split()**
    - **strip()**
    - **swapcase()**
    - **upper()**
        - convert all the chraracters to upper
    - **zfill()**

file I/O

- file
    - collection of data
    - a way to persist the data/information
    - data will be stored in secondary storage with (absolute or relative path)
- file modes
    - operation
        - **read mode (r)**
            - used to read file (write is not possible)
            - default mode
        - **write mode (w)**
            - used to write file
            - new file will be created if the file does not exist on the path
        - **append mode (a)**
            - used to append new data to existing contents
    - file format
        - t
            - text file
            - default format
        - b
            - binary file
- operations
    - **open()**
        - used to open a file
    - **close()**
        - used to close a file
    - **read()**
        - used to read whole file
    - **read(n)**
        - used to read n bytes (from the filter pointer location)
    - **readlines()**
    - **write()**
        - used to write data (string) into a file
    - **seek()**

        - used to set the file pointer to point to a specific location
    - **tell()**
        - returns the position the file pointer is point to

built-in

- **print()**: used to print something on console
- **type()**: used to get data type of a variable
- **range()**
    - used to get sequential values
    - e.g.

```
numbers_1 = list(range(0, 10))

# numbers_1 = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(f"numbers_1 = {numbers_1}")

numbers_2 = list(range(0, 10, 2))

# numbers_2 = [0, 2, 4, 6, 8]
print(f"numbers_2 = {numbers_2}")

numbers_3 = list(range(1, 10, 2))

# numbers_3 = [1, 3, 5, 7, 9]
print(f"numbers_3 = {numbers_3}")

# if the start is missing, the range will start from 0
numbers_4 = list(range(10))

# numbers_4 = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(f"numbers_4 = {numbers_4}")
```

Type casting

- to int: int()
- to float: float()
- to string: str()
- to bool: bool()
- to list: list()
- to tuple: tuple()

collections

- similar to array in other languages

- group of values

list

- collection of similar or dissimilar values
- **array**: collection of similar values
- list is **mutable**: once created, a list CAN be modified
- list is always slower than tuple
- allows duplicate values
- to create a list
    - use []

```
# empty list
list_1 = []

# list
print(f"type = {type(list_1)}")
```

    - call list()

```
# empty list
list_1 = list()

# list
print(f"type = {type(list_1)}")
```

- operations
    - **len()**
        - used to get the number of values inside a list
        - e.g.

```
numbers = [10, 20, 30, 40, 50, 60, 70]

# number of values in numbers = 7
print(f"number of values in numbers = {len(numbers)}")
```

- **append()**
  - used to add new value to the end of the collection
  - e.g.

```
numbers = [10, 20, 30, 40, 50]

# [10, 20, 30, 40, 50, 60]
numbers.append(60)

# [10, 20, 30, 40, 50, 60, 70]
numbers.append(70)
```

- **insert(index, value)**
  - used to add a value at an index position
  - e.g.

```
numbers = [10, 20, 30, 40, 50]

# [10, 20, 100, 30, 40, 50]
numbers.insert(2, 100)
```

- **pop()**
  - used to remove the last value from the collection
  - e.g.

```
numbers = [10, 20, 30, 40, 50]

# [10, 20, 30, 40]
numbers.pop()

# [10, 20, 30]
numbers.pop()
```

- **pop(index)**
  - used to remove the value at the index from the collection

- e.g.

```
countries = ["india", "usa", "uk", "china", "japan"]

# ["india", "usa", "uk", "japan"]
countries.pop(3)
```

- **remove(value)**
  - used to remove the value from the collection
  - e.g.

```
countries = ["india", "usa", "uk", "china", "japan"]

# ["india", "usa", "uk", "japan"]
countries.remove('china')
```

- **index()**
  - used to find the position of a value
  - by default index() searches the value from 0th position
  - e.g.

```
numbers = [10, 20, 30, 40, 60, 70, 80, 40, 90, 100, 40, 110,
40]

# index of 40 = 3
print(f"index of 40 = {numbers.index(40, 0)}")

# index of 40 = 7
print(f"index of 40 = {numbers.index(40, 4)}")

# index of 40 = 10
print(f"index of 40 = {numbers.index(40, 8)}")

# index of 40 = 12
print(f"index of 40 = {numbers.index(40, 11)}")
```

- **count(value)**
  - used to find the number of occurances of a value

- e.g.

```
numbers = [10, 20, 30, 40, 60, 70, 80, 40, 90, 100, 40, 110,
40]

# 40 is repeated 4 times
print(f"40 is repeated {numbers.count(40)} times")
```

- **sort()**
  - used to sort the list
  - e.g.

```
numbers = [10, 2, 3, 1, 8, 3, 4, 6, 5]

# Ascending
# [1, 2, 3, 3, 4, 5, 6, 8, 10]
numbers.sort()

# Descending
# [10, 8, 6, 5, 4, 3, 3, 2, 1]
numbers.sort(reverse=True)
```

- **reverse()**
  - used to reverse a list
  - e.g.

```
countries = ["india", "usa", "uk", "japan"]

# ['japan', 'uk', 'usa', 'india']
countries.reverse()
```

- **copy()**
  - used to create a new copy of existing list
  - e.g.

```
numbers = [10, 30, 50, 40, 20]
```

```
# make a copy of numbers
numbers_1 = numbers.copy()

# sort the newly created copy
# this WILL NOT modify the original numbers
numbers_1.sort()

# [10, 30, 50, 40, 20]
print(numbers)

# [10, 20, 30, 40, 50]
print(numbers_1)
```

- **clear()**
  - used to remove all the values from a list
  - e.g.

```
numbers = [10, 30, 50, 40, 20]

numbers.clear()

# []
print(numbers)
```

- **extend()**
  - used to add values from one list to another
  - e.g.

```
numbers_1 = [10, 20, 30, 40, 50]
numbers_2 = [60, 70, 80, 90, 100]

numbers_1.extend(numbers_2)

# [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
print(numbers_1)
```

Indexing

- way to retrieve value(s) from the collection

- types
    - positive
        - starts from left
        - the element will have a position = 0
        - e.g.

```
numbers = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]

# numbers[0] = 10
print(f"numbers[0] = {numbers[0]}")

# numbers[9] = 100
print(f"numbers[9] = {numbers[9]}")
```

    - negative
        - starts from right
        - the last element will have a position = -1
        - e.g.

```
numbers = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]

# numbers[-10] = 10
print(f"numbers[-10] = {numbers[-10]}")

# numbers[-1] = 100
print(f"numbers[-1] = {numbers[-1]}")
```

Slicing

- getting a part/portion of collection
- syntax:
    - <collection> [<start> : <stop>]
    - <collection> [<start> : <stop> : <step_count>]
- rules
    - stop must be greater than start
    - both of them are optional
        - if start is missing, the slicing will start from 0
        - if stop is missing, the slicing will stop at the last value in the collection
- e.g.

```
numbers = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]

# numbers[3:7]  = [40, 50, 60, 70]
print(f"numbers[3:7]  = {numbers[3:7]}")

# numbers[0:5]  = [10, 20, 30, 40, 50]
print(f"numbers[0:5]  = {numbers[0:5]}")

# numbers[:5]   = [10, 20, 30, 40, 50]
print(f"numbers[:5]   = {numbers[:5]}")

# numbers[6:10] = [70, 80, 90, 100]
print(f"numbers[6:10] = {numbers[6:10]}")

# numbers[6:]   = [70, 80, 90, 100]
print(f"numbers[6:]   = {numbers[6:]}")

# numbers[:]   = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
# numbers[0:10]
print(f"numbers[:]   = {numbers[:]}")
```

```
numbers = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]

# numbers[0:9:2]  = [10, 30, 50, 70, 90]
print(f"numbers[0:9:2]  = {numbers[0:9:2]}")

# numbers[1:9:2]  = [20, 40, 60, 80]

print(f"numbers[1:9:2]  = {numbers[1:9:2]}")

# numbers[::]  = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
print(f"numbers[::]  = {numbers[::]}")

# numbers[::]  = [100, 90, 80, 70, 60, 50, 40, 30, 20, 10]
print(f"numbers[::]  = {numbers[::-1]}")
```

tuple

- collection of similar or dis-similar values
- use () to create a tuple
- tuple is immutable: once created, tuple CAN NOT be modified (read-only)
- tuple is always faster than list

- allows duplicate values
- to create an empty tuple
    - use ()

```
# empty tuple
tuple_2 = ()
print(tuple_2)
```

    - use tuple()

```
# empty tuple
tuple_1 = tuple()
print(tuple_1)
```

- tuple with one value

```
# integer variable
tuple_1 = (10)

# tuple with one value
tuple_2 = (10,)

 # string variable
tuple_3 = ("test")

# tuple with one value
tuple_4 = ("test",)
```

- operations
    - **index**
        - used to find the first index position of the value
    - **count**
        - used to find number of occurences of a value

list vs tuple

list	tuple

list	tuple
1. mutable	1. immutable
2. use []	2. use ()
3. list with one value [10]	3. tuple with one value (10,)

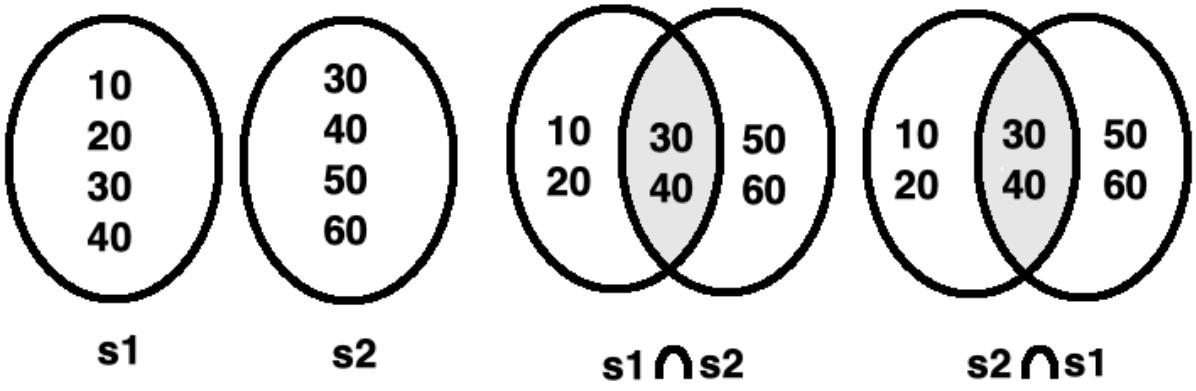
set

- collection of unique values (does not allow duplicates)
- does NOT follow the insertion order
- is mutable: once created, set can be modified
- to create an empty set
    - call function set()

```
# it is an empty dictionary
# c3 = {}

# empty set
c3 = set()
```

- operations
    - **intersection**
        - getting only common values between two sets
        - e.g.

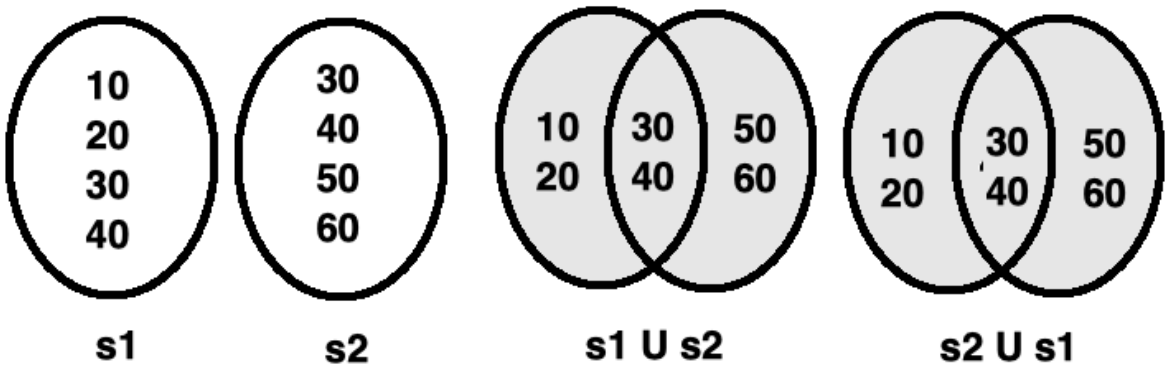



```
s1 = {10, 20, 30, 40}
s2 = {30, 40, 50, 60}
```

```
# {30, 40}
print(s1.intersection(s2))

# {30, 40}
print(s2.intersection(s1))
```

- **union**
  - combining two sets by keeping common elements only once
  - e.g.

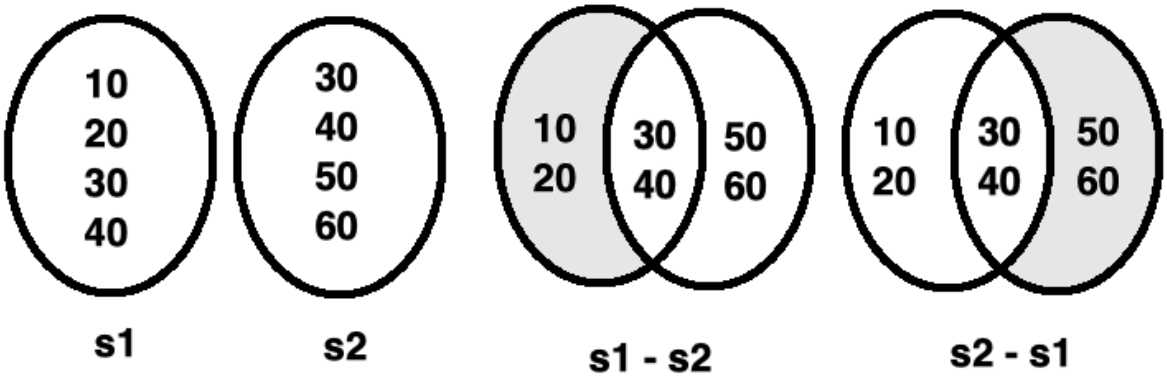



```
s1 = {10, 20, 30, 40}
s2 = {30, 40, 50, 60}

# {10, 20, 30, 40, 50, 60}
print(s1.union(s2))

# {10, 20, 30, 40, 50, 60}
print(s2.union(s1))
```

- **difference**
  - select only uncommon elements from first operand
  - difference is not a symmetric operation
  - e.g.




```
s1 = {10, 20, 30, 40}
s2 = {30, 40, 50, 60}

# {10, 20}
print(s1.difference(s2))

# {50, 60}
print(s2.difference(s1))
```

frozenset

- immutable set

dictionary

- collection of key-value pairs
- to create empty dictionary
    - use {}

```
d1 = {}
```

    - use dict()

```
d1 = dict()
```

- dictionary with key-value pairs

```
person = {
    "email": "person1@test.com",
    "address": "pune",
    "name": "person1",
    "age": 40
}
```

- operations
    - **keys**
        - returns list of keys

```
person = {
  "email": "person1@test.com",
  "address": "pune"
}

# ["email", "address"]
print(person.keys())
```

    - **values**
        - returns list of values

```
person = {
  "email": "person1@test.com",
  "address": "pune"
}

# ["person1@test.com", "pune"]
print(person.values())
```

    - **pop(key)**
        - used to removes a key-value pair from a dictionary
        - e.g.

```
person = {
```

```
    "name": "person1",
    "age": 40
}

# {"name": "person1", "age": 40}
print(person)

person.pop("age")

# {"name": "person1"}
print(person)
```

- **get(key)**
  - used to retrieve value of a key
  - get return None if the key is missing in the dictionary [application does not crash]
  - e.g.

```
person = {
    "name": "person1",
    "age": 40
}

# name = "person1"
print(f"name: {person.get('name')}")

# phone = None
print(f"phone: {person.get('phone')}")

# the application crashes by sending an error (KeyError)
print(f"phone: {person['phone']}")
```

- **items()**
  - used to return the key-value pairs
  - returns list of tuples (key, value)
  - e.g.

```
person = {
    "name": "person1",
    "age": 40
}

# [('name', 'person1'), ('age', 40)]
```

```
    print(person.items())
```

loops

for..in

- used to iterate over iterable (collections)
- e.g.

```
for value in range(5):
    print(f"value = {value}")
```

for..in..else

- else block in for..in loop gets called only when the for loop does not break
- e.g.

```
# the else block will get called
for value in range(5):
    print(f"value = {value}")
else:
    print("this is for's else block")

# the else block will NOT get called
for value in range(5):
    print(f"value = {value}")
    if value > 3:
      break
else:
    print("this is for's else block")
```

while

closure

decorator

---

object oriented programming

terminology

class

- collection of
  - attributes
    - characteristic of an entity
    - e.g. name of person, address of person, model of car
  - methods
    - function inside a class
- blueprint to create an object
- e.g.

```
# empty class
class Person:
  pass
```

object

- instance of a class
- memory allocated to store the class attributes
- charactertistics
  - every object must be identified by a unique address
  - state:
    - the value of every attribute at a give time

instantiation

- process of creating an object
- e.g.

```
class Person:
  pass

# person1 is called as reference
# person1 will refer an object of type Person
person1 = Person()
```

method

- function declared inside the class
- types

- **initializer**
  - method which initializes the object
  - gets called automatically/implicitly
  - **NOTE**:
    - never call __init__() explicitly
  - gets called for number of objects created from the class
  - must have a name = **__init__**
  - types
    - default
      - also known as parameterless initializer
      - e.g.

```
class Person:

  # default initializer
  def __init__(self):
    print("inside __init__")
```

    - custom
      - also known as parameterized initializer
      - e.g.

```
class Person:

  # default initializer
  def __init__(self, name, age):
    print("inside __init__")
    self.name = name
    self.age = age
```

- **de-initalizer (delete)**
  - used to de-initialize the object
  - gets called automatically/implicitly
  - **NOTE**:
  - never call __del__() explicitly

- gets called for number of objects
- e.g.

```
class Person:

  # initializer
  def __init__(self):
    print("inside __init__")

  def __del__(self):
    print("inside __del__")
```

- **setter**
  - used to set a new value to an attribute
  - also known as mutator
  - e.g.

```
class Person:

  # initializer
  def __init__(self, name):
    self.__name = name

  # setter
  def set_name(self, name):
    self.__name = name
```

- **getter**
  - used to get a value of an attribute
  - also known as inspector
  - e.g.

```
class Person:

  # initializer
  def __init__(self, name):
    self.__name = name

  # getter
  def get_name(self, name):
```

```
        return self.__name
```

- **facilitator**
  - adds facility in the class
  - e.g.

```
class Person:

  # initializer
  def __init__(self, name, age):
    self.name = name
    self.age = age

  # facilitator
  def can_vote(self):
    if self.age >= 18:
      print("yes")
    else:
      print("no")
```

access specifiers

- specify the access level of an attribute/method
- types
  - **public**
    - are accessible outside the class
    - any member declared without using any underscore is treated as public member
    - e.g.

```
class Car:
  def __init__(self, model, price):

    # public memebers
    self.model = model
    self.price = price

car = Car('i20', 7.5)

# we can access/modify the public members
car.model = 'new model'
```

```
car.price = 10.6
```

- **protected**
  - are accessible in the same class and all of its child classes
  - use underscore (_) as a prefix
  - e.g.

```
class Vehicle:
  def __init__(self, engine):

    # protected memebers
    self._engine = engine

class Car:
  def __init__(self, model, price, engine):
    Vehicle.__init__(self, engine)

    # protected memebers
    self._model = model
    self._price = price

  def print_info():
    # protected member is accessible in child class
    print(f"engine: {self._engine}")
```

- **private**
  - are accessible only inside the class
  - are NOT accessible outside the class
  - members which start with __ are treated as private members
  - e.g.

```
class Car:
  def __init__(self, model, price):

    # private memebers
    self.__model = model
    self.__price = price

car = Car('i20', 7.5)

# we can NOT access/modify the public members
# car.__model = 'new model'
```

```
    # car.__price = 10.6
```

code reuse

- resuing the code

association

- represents associations of multiple classes
- types
    - aggregation
        - also known as has-a relationship
        - loose coupling / weak relationship
        - one entity can live without another entity
        - one object contains an object of another class
        - e.g.
            - Student has-a address
            - Company has-a address
            - Person has-a name
            - Company has-a employee
            - Department has-a professor
    - composition
        - also known as composed-of / part-of
        - tight coupling / strong relationship
        - one entity can not live without the other entity
        - e.g.
            - Car composed-of engine
            - Room composed-of wall
            - human composed-of heart

inheritance

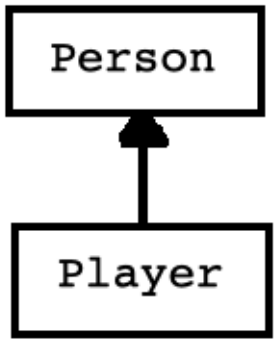
- also known as is-a relationship
- one class is made up of another class
- the base class can be specified at the time of declaration with ()
- e.g.
    - player is-a person
    - bike is-a vehicle
    - lion is-a animal

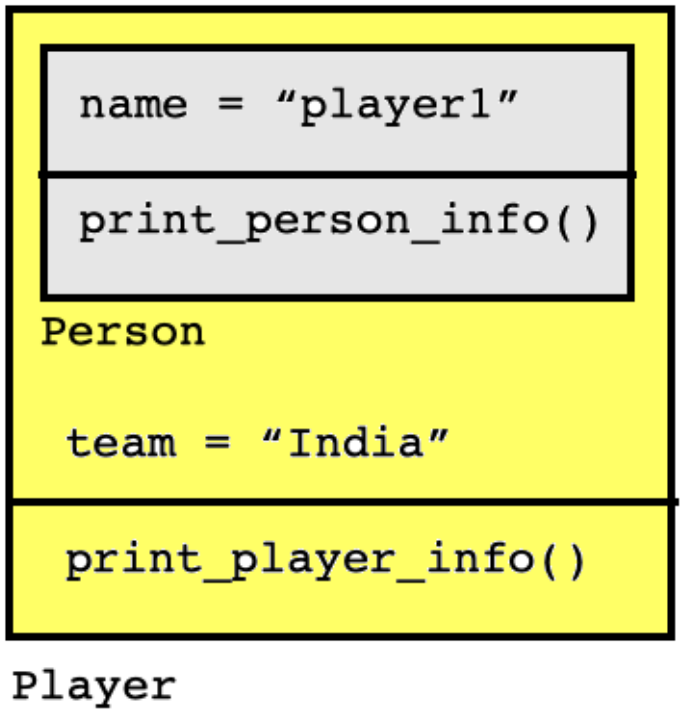
```
    # Base class
```

```
class Person:
  pass

# Derived class
class Employee(Person):
  pass
```

- types
  - **single inheritance**
    - there is only one base class and only one derived class
    - e.g.





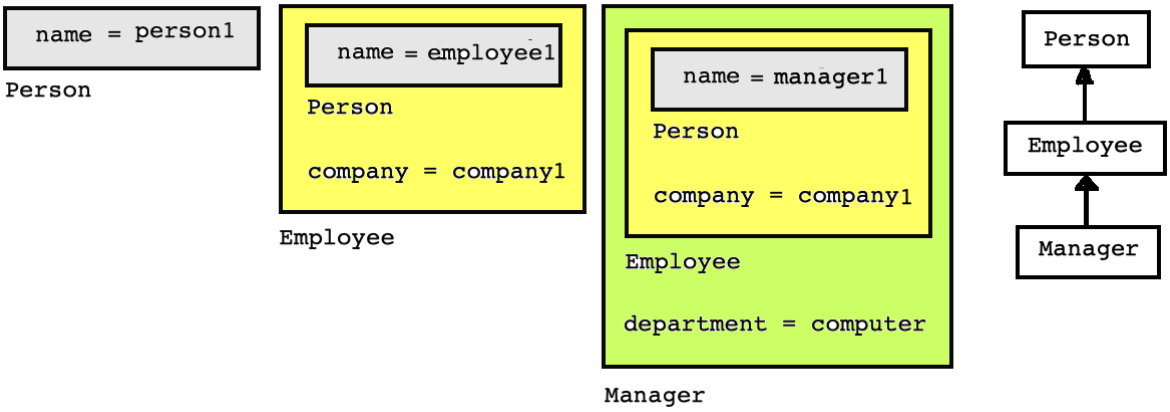

```
# base class
class Person:
    def __init__(self, name):
        self.name = name

    def print_person_info(self):
        print(f"name = {self.name}")

# dervied class
class Player(Person):
    def __init__(self, name, team):
        Person.__init__(self, name)
        self.team = team
```

```
        def print_player_info(self):
            print(f"name = {self.name}")
            print(f"team = {self.team}")
```

- **multi-level**
  - there are multiple levels
  - a level will have one base and one derived class
  - a may have one direct and mulitple indirect base class(es)
  - e.g.

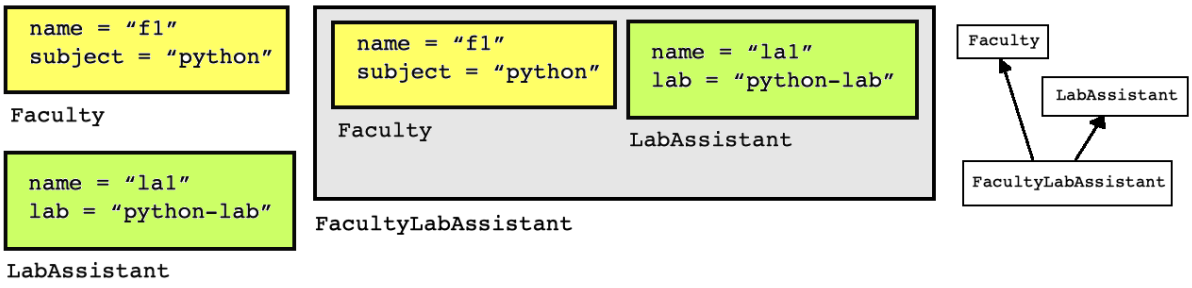



```
class Person:
    def __init__(self, name):
        self._name = name


class Employee(Person):
    def __init__(self, name, company):
        Person.__init__(self, name)
        self._company = company


class Manager(Employee):
    def __init__(self, name, company, department):
        Employee.__init__(self, name, company)
        self._department = department
```

- **multiple**
  - multiple base classes
  - one derived class
  - e.g.

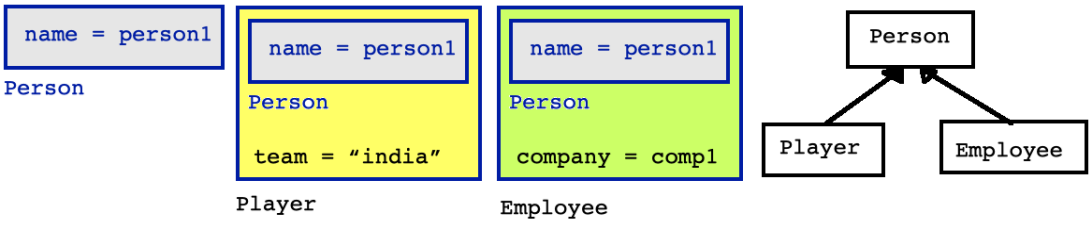



```
class Faculty:
    def __init__(self, name, subject):
        self._name = name
        self._subject = subject


class LabAssistant:
    def __init__(self, name, lab):
        self._name = name
        self._lab = lab


class FacultyLabAssistant(Faculty, LabAssistant):
    def __init__(self, name, subject, lab):
        Faculty.__init__(self, name, subject)
        LabAssistant.__init__(self, name, lab)
```

- **hierarchical**
  - one base class and multiple derived classes
  - e.g.

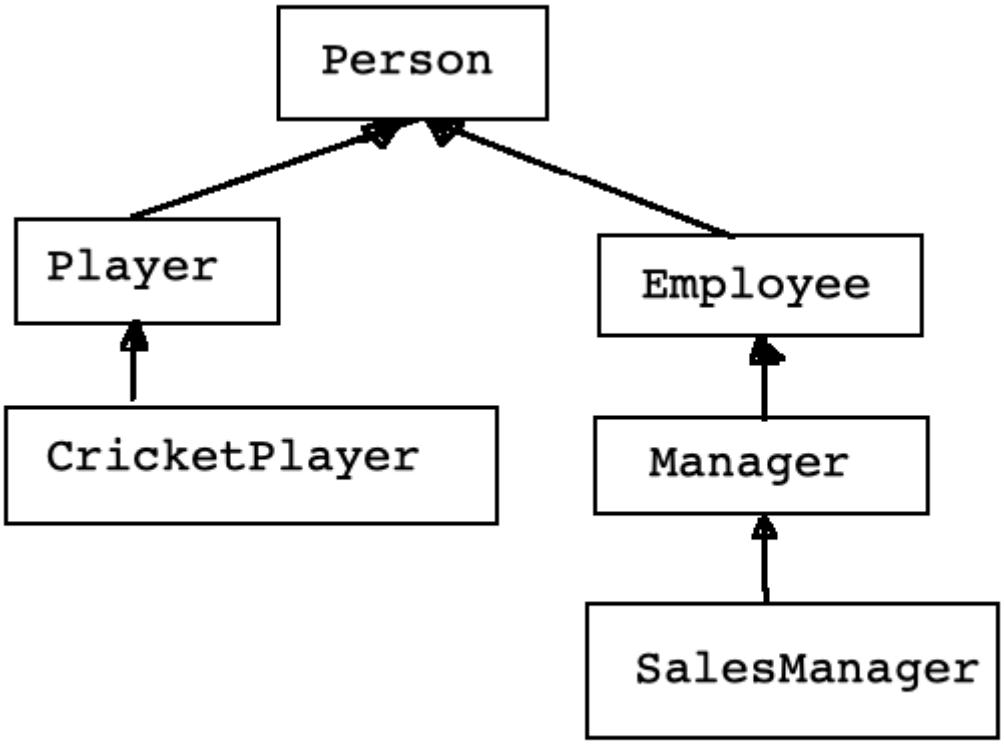



```
class Person:
    def __init__(self, name):
        self._name = name


class Player(Person):
    def __init__(self, name, team):
        Person.__init__(self, name)
        self._team = team
```

```
class Employee(Person):
    def __init__(self, name, company):
        Person.__init__(self, name)
        self._company = company
```

- **hybrid**
  - combination of any two or more inheritance types is known as hybrid
  - e.g.




```
class Person:
    def __init__(self, name):
        self._name = name


class Player(Person):
    def __init__(self, name, team, sport):
        Person.__init__(self, name)
        self._team = team
        self._sport = sport
```

```
class CricketPlayer(Player):
    def __init__(self, name, team):
        Player.__init__(self, name, team, 'Cricket')


class Employee(Person):
    def __init__(self, name, company):
        Person.__init__(self, name)
        self._company = company


class Manager(Employee):
    def __init__(self, name, company, department):
        Employee.__init__(self, name, company)
        self._department = department


class SalesManager(Manager):
    def __init__(self, name, company, department, target):
        Manager.__init__(self, name, company, department)
        self._target = target
```

NOTE

- Base class CAN NOT access any member(s) of derived class
- Derived class CAN access any (protected or public) member(s) of base class(es)

```
class Person:
  def print_person():
    pass

class Student(Person):
  def print_student():
    pass

p1 = Person()

# p1 can call any method of Person
p1.print_person()

# p1 can not access any method of Student
# Person is a base class of Student
# p1.print_student()

s1 = Student()

# s1 can call any method of Student
s1.print_student()
```

```
# s2 can calll any method of Person
# Student is a derived class of Person
s1.print_person()
```

root class

- in python (3.0 +), object is a root class
- in python, every class is a derived class of object (directly or indirectly)
- object class provides basic functionality
    - converting any object to string (**str**)

method overriding

- in inheritance scenario, when derived class uses method with same name as that of the base class
- used to change the behavior/implementation of base class method
- to overide a method
    - in derived class, use same name of the method as that of the base class
- e.g.

```
class Vehicle:
  def __init__(self, engine):
    self._engine = engine

  def print_info(self):
    print(f"engine: {self._engine}")

class Car(Vehicle):
  def __init__(self, engine, model):
    Vehicle.__init__(self, engine)
    self._model = model

  # Car class is overiding the print_info method
  def print_info(self):
    Vehicle.print_info(self)
    print(f"model: {self._model}")
```

operator overloading

- changing the default behavior of built-in operators
- **comparison operators**
    - p1 > p2 : **__gt__**
    - p1 < p2 : **__lt__**

    - p1 <= p2: **__le__**
    - p1 == p2: **__eq__**
    - p1 != p2: **__ne__**
    - p1 >= 2 : **__ge__**

- **mathematical operators**

    - p1 + p2 : **__add__**
    - p1 – p2 : **__sub__**
    - p1 * p2 : **__mul__**
    - p1 / p2 : **__truediv__**
    - p1 // p2: **__floordiv__**
    - p1 ** 2 : **__pow__**
    - p1 % p2 : **__mod__**

exception handling

- exception
    - run time condition because of which the application may crash
    - types
        - exception
            - the one which can be handled
            - e.g. file not found, divide by zero
        - error
            - the one which can NOT be handled
            - e.g. no memory
- exception handling
    - handling an exception
    - blocks
        - try
            - contains the code which may raise an exception
            - python will try to execute the code and if it generates any exception, except block gets called
        - except
            - responsible for handling the exception
            - responsible for taking an action(s) when an exception is raised
            - will be called when one ore more error(s) are raised
        - else
            - will be called when no exception is raised
        - finally
            - will be called irrespective of any exception being raised
- e.g.

```
try:
  #
  # piece of code which may raise an exception
```

```
    #
except Exception:
    # handle the exception
else:
    # block will be called when there is no exception raised
finally:
    # this block will always get executed
```

module

- file with .py extension
- collection of
    - variables
    - functions
    - classes
- __name__ is used to get the name of a module
- __main__ is the module name of currently executing module
- e.g.

```
# page: page1.py
num = 100

# the file with nane page1.py is representing a module with name page1
```

- importing a module

```
# mymodule.py
class Person:
    pass

def my_function():
    pass


# page1.py
# import all the entities from mymodule
import mymodule
p1 = mymodule.Person()


# page2.py
# import only Person class from mymodule
from mymodule import Person
p1 = Person()
```

```
# page3.py
# MyPerson will be created as an alias of Person
from mymodule import Person as MyPerson
p1 = MyPerson()


# page4.py
# mm will be created as an alias for mymodule
import mymodule as mm
p1 = mm.Person()
```

package

- collection of modules
- folder with a file named __init__.py
- e.g.

```
# mypackage           (package)
# - files             (sub-package)
#   - myfile          (module)
#     - Person        (class)
#     - Car           (class)

# from mypackage.files.myfile import Person
# p1 = Person()

# from mypackage.files.myfile import Car
# c1 = Car()
```