**Python**

* Language
  + medium for expressing the ideas (programs)
  + programs => implementations of algorithms
  + algorithm => set of instruction
  + types based on the level
    - low level language (ASM)
    - middle level: (C, C++)
    - high level: (Java, python)
  + types based on the translation
    - compiled
      * which uses compiler to convert the source code to ASM code
      * compiler:
        + tools which converts source code to ASM code
        + the executable will be created for execution
      * used for developing native applications
      * E.g. C, C++
      * advantages
        + faster than interpreted language
      * disadvantages
        + executables are platform/OS specific
    - interpreted
      * which uses interpreter to execute the source code
      * interpreter
        + tools which translates the source code into ASM code line by line and produces the output
      * E.g. python, PHP, Perl, Ruby, JavaScript, HTML
      * advantages
        + platform/OS independent
      * disadvantage
        + slower than compiled/native applications
    - hybrid
      * shows both compiled and interpreted features
      * E.g. Java
* overview of python
  + Guido Rossum has started Python in 1991
  + interpreted language
  + general purpose language
    - console application
    - GUI application
    - Web Application
    - scripts
    - data processing
  + versions
    - v2
      * last version: 2.7.6
    - v3
      * latest version: 3.5 -> 3.6 -> 3.7
  + features
    - supports dynamic typing
      * the data type will be assigned to the variables at the time of execution
    - is
      * scripting language
      * Object Oriented language
      * functional language
    - supports automatic memory management (garbage collection)
    - does not support pointers
* python installation
  + Linux:
    - by default: 2.x
    - ubuntu/debian
      * sudo apt-get install python3 pip3
    - Suse/Red Hat/Centos
      * sudo yum install python3 pip3
  + Windows:
    - download the python installer
  + macOS:
    - by default: python 2.6
    - brew install python3 pip3
* Editor
  + vim
  + Visual Studio Code
  + IDE: Integrated Development Environment
    - Spyder (anaconda)
    - Pycharm
* python fundamentals
  + identifier
    - entity which can identify variable, function, class, keyword etc
    - python provides at least 33 keywords
    - rules
      * CAN NOT use special characters like space
        + e.g. first name = “”: invalid
        + first\_name = “”: valid
        + firstName = “”: valid
  + variable
    - to declare a variable
      * syntax:
        + <var name> = <initial value>
    - every variable requires the initial value
  + keywords:
    - def: used to define a function
    - pass:
      * placeholder. does not do anything
      * is used to define a empty function
    - None
      * nothing
    - True
    - False
    - global:
      * is used to specify the variable’s global scope
      * e.g.
        + num = 100
        + def function1():

global num

num = 200

print(num) # 200

* + - * + print(num) # 200
        + function1()
    - nonlocal:
      * is used to modify a variable defined by outer function
      * force inner function not to declare a local variable
      * e.g.
        + def outer():

num = 100

def inner():

nonlocal num

print(num) # 100

num = 200

print(num) # 200

inner()

print(num) # 200

* + - lambda:
      * a kind of anonymous function (unnamed function)
      * similar to inline functions in C
      * faster than named function
      * always prefer lambda over a named function if possible
      * syntax
        + lambda <parameter(s)>: <body>
      * rules
        + must contain only one statement in the body section
        + the statement in body section must return a value
        + if the explicit return is not there then the statements final value will be returned by default
        + may accept one or more parameters
        + must accept at least one parameter
      * e.g.
        + square = lambda x: x \*x
        + add = lambda x, y: x + y
        + print(square(10)) # 100
        + print(add(10, 20)) # 30
    - is:
      * used to check if two references are referring to the same object
      * e.g.
        + class A:

pass

* + - * + a1 = A()
        + a2 = A()
        + a3 = a1
        + print(a1 is a2) # False
        + print(a1 is a3) # True
    - in:
      * used to check if a value is present in a collection
      * e.g.
        + numbers = [1, 2, 3, 4, 5]
        + print(4 in numbers) # True
        + print(30 in numbers) # False
* statements
  + basic unit of execution
  + rule
    - semi-colon (;) is not required when a statement is written on one line
    - but when multiple statements are written on the same line then semi-colon (;) is mandatory
  + types
    - comment
      * use # symbol to create a comment
    - assignment statement
      * e.g. num = 10
    - control statements
      * if..else
        + e.g.

if age >= 18:

print(“Yes”)

else:

print(“No”)

* + - * if..elif..else
        + e.g.

if marks < 90:

print(“A”)

elif marks < 70:

print(“B”)

else:

print(“F”)

* block
  + collection of statements
  + types
    - function
    - if..else
    - if..elif..else
    - for..in
    - do..while
* data types
  + all the data types are **inferred**
    - data type will be automatically decided by python
  + types
    - int: whole number
    - float: decimal number
    - string (str):
      * list of characters
      * can be declared with
        + single quote (‘): used to declare single line value

e.g. name = “steve jobs”

* + - * + double quote (“): used to declare single line value

e.g. name = ‘steve jobs’

* + - * + tripe double quotes (“””): used to declare multi line value

e.g. address = “””

house no 100,

city

“””

* + - * e.g.
        + dialog = "arnold once said, \"Trust me!! I will be back\""
        + dialog = 'arnold once said, "Trust me!! I will be back"'
        + dialog = 'arnold once said, \'Trust me!! I will be back\''
        + dialog = "arnold once said, 'Trust me!! I will be back'"
      * methods:
        + lower(): used to convert all the characters to lower case
        + upper(): used to convert all the characters to upper case
        + title(): used to convert the value to title case
        + join():

used to convert a list to a string

e.g.

l1 = [‘a’, ‘b’, ‘c’]

print(‘’.join(l1)) # ‘abc’

print(‘\*’.join(l1)) # ‘a\*b\*c’

* + - * + format():

used to format a string

e.g.

name = “test”

# name = ‘test’

print(“name = {}”.format(name))

* + - boolean (bool):
      * uses one of the **True** or **False** values
      * e.g.
        + canVote = True
        + canVote = False
    - object
* collection
  + group of objects
  + types
    - list ([])
      * collection of similar or dissimilar objects/values
      * allows duplicate values
      * mutable in nature
        + list object can be modified at run time

new value can be added

existing value can be deleted/update

* + - * to create a list
        + use [] like

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

countries = [“India”, “USA”]

* + - * + use list() like

numbers = list(range(0, 5)) # [0, 1, 2, 3, 4]

* + - * methods
        + append(): used to append a new value at the end of the list
        + pop(): used to remove the last value
        + insert(): used to add a value at a specific index
        + remove(): used to remove a specific value
        + sort(): used to sort the list
        + reverse(): used to reverse the list
    - tuple (())
      * immutable in nature
        + once created tuple can not be updated
      * always prefer tuple over list
      * tuple is faster than list
        + accessing tuple elements is faster than accessing list elements
    - set ({})
      * does not allow duplicate values
      * does not follow insertion order
      * to create mutable set
        + use {}
        + use set()
      * to create immutable set
        + use frozenset()
      * methods
        + intersect():
        + union():
      * operation:
        + s1 – s2
    - dictionary ({})
      * collection of key: value pairs
      * e.g.
        + person = {

‘name’: ‘steve’,

‘address’: ‘usa’,

‘age’: 58

* + - * + }
      * where
        + name, address, age: keys
        + steve, usa, 58: values

|  |  |
| --- | --- |
| **List** | **Tuple** |
| 1. mutable | 1. immutable |
| 1. slower than tuple | 1. is faster than list |

|  |  |
| --- | --- |
| **List** | **Set** |
| * + - allows duplicate values | * + - does not allow duplicate |
| * + - follows insertion order | * + - does not follow insertion order |

* + dimensionality
    - one dimensional
    - multi-dimensional
      * collection of collections
* operators
* loops
* functions
  + block of statements with a name
  + every function in python returns a value
    - if developer returns a valid value then the value becomes the return value
    - by default the function returns None
  + system functions
    - print(): used to print something on console
    - type(): used to print the data type of a variable
    - range():
      * used to create range from start to end with step values
      * param1: start, param2: end, param3: step
      * end value will be always excluded from the list
    - list(): used to create a list
    - len(): used to get length of a collection
    - int(): used to convert. into int
    - float(): used to convert. into float
    - str(): used to convert. into string
    - map():
      * used to process every value in the collection
      * accepts
        + function name / address of a function
        + collection
      * syntax:
        + map(<function name>, <collection>)
        + where

function must return a value

* + - * e.g.
        + numbers = [ 1, 2, 3, 4, 5 ]
        + # [1, 4, 9, 16, 25]
        + squares = list(map(lambda x: x \* x, numbers))
    - filter():
      * used to filter the collection
      * it works with only Boolean values (True/False) returned from the function
      * syntax:
        + filter(<function>, <collection>)
        + where

function must return True/False

* + - * e.g.
        + numbers = [1, 2, 3, 4, 5]
        + # [2, 4]
        + even = list(filter(lambda x: x % 2 == 0, numbers))
    - setattr(): used to set a value to an attribute inside a class
    - getattr(): used to get a value of an attribute from a class
    - isinstance():
      * used to check if an object/instance is of a specific type
      * e.g.
        + class A:

pass

* + - * + class B(A):

pass

* + - * + class C(B):

pass

* + - * + a = A()
        + b = B()
        + c = C()
        + print(isinstance(a, A)) # True
        + print(isinstance(b, B)) # True
        + print(isinstance(b, A)) # True
        + print(isinstance(c, C)) # True
        + print(isinstance(c, A)) # True
        + print(isinstance(c, B)) # True
        + print(isinstance(a, B)) # False
        + print(isinstance(a, C)) # False
        + print(isinstance(b, C)) # False
  + custom functions / user defined functions
    - syntax:

**def** <function name>():

<indentation>”””

<indentation> <documentation string>

<indentation>”””

<indentation> <code>

* + - e.g.
      * parameterless function

def function1():

“””

this is a test function

“””

print(“inside function1”)

* + - to print the docstring of a function use \_\_doc\_\_ property

print(function1.\_\_doc\_\_) # this is a test function

* + features
    - default parameter value/optional parameter
      * a function while declaration can assign default value(s) to parameter(s)
      * if a parameter gets a default value, then caller can skip passing the value for the parameter
      * the parameter having default value becomes optional
      * all the parameters having default values must be towards end of the parameters list
      * e.g.
        + def function1(p1 = 10):

pass

* + - * + function1() # p1 = 10
        + function1(200) # p1 = 200
    - named parameters
      * caller may call the function by passing the values along with the parameter name
      * e.g.
        + def function(p1, p2):

pass

* + - * + function(10, 20) # p1: 10, p2: 20
        + function(p2=20, p1=10) # p1: 10, p2: 20
      * e.g.
        + def function(p1, p2, p3=30):

pass

* + - * + function(10, 20) # p1: 10, p2: 20, p3: 30
        + function(p2=20, p1=10) # p1: 10, p2: 20, p3: 30
        + function(p3=100, p2=10, p1=20) # p1: 20, p2: 10, p3: 100
    - variable length argument function
      * function which accepts variable length arguments
      * all the values will be added into a tuple and the tuple will be passed to the function as a parameter
      * e.g.
        + def add(\*arguments):

# arguments is a tuple

print(arguments)

* + - * + add(1, 2) # (1, 2)
        + add(1, 2, 3) # (1, 2, 3)
        + add(1, 2, 3, 4) # (1, 2, 3, 4)
  + types
    - Empty function
      * without having a body
      * e.g.
        + def function():

pass

* + - Nested function
      * a function within another function
      * a inside function is called as inner function
      * a function which contains inner function is called as an outer function
      * syntax:
        + def outer():

print(‘inside outer’)

def inner():

print(‘inside inner’)

inner()

* + - * feature
        + inner function can be called only within outer function
        + inner function can access all the local entities defined by outer function
* scope
  + local
    - defined inside a function
    - e.g.
      * def function1():
        + num = 200 # local
        + print(num) # 200
  + global
    - defined outside a function
    - e.g.
      * num = 100 # global
      * print(num) # 100
      * def function1():
        + print(num) # 100
      * print(num) # 100
  + outer
    - entity is declared by outer function
  + rule
    - if a function wants to modify the value of a global variable, the function needs to declare the variable with global keyword
    - e.g.
      * num = 100
      * print(num) # 100
      * def function1():
        + global num
        + num = 200
        + print(num) # 200
      * print(num) # 200
* List comprehension
  + way to iterate over a collection
  + reduces code size
  + as a combination of map and filter
  + syntax:
    - [<return value> <for in> <condition>]
    - where
      * if condition is not present then it works like map()
      * if condition is present then it works like filter()
  + e.g.
    - numbers = [1, 2, 3, 4, 5]
    - [n \* n for n in numbers] # [1, 4, 9, 16, 25]
    - [n for n in numbers if n % 2 == 0] # [2, 4]
    - [(n, n\*n) for n in numbers] # [(1, 1), (2, 4) …]
  + limitations
    - CAN NOT contain nested for..in loops
    - CAN NOT contain else condition
    - the if condition must return True / False
    - for..in loop CAN NOT contain more than one statement
* Closure
  + a way to call inner function outside of outer function
  + syntax:
    - def <outer function>():
      * def <inner function>():
        + <inner body>
      * return <inner function>
  + e.g.
    - def createTag(tag):
      * def inner(data):
        + print(‘<{}>{}</{}>’.format(tag, data, tag))
      * return inner
    - p = createTag(‘p’)
    - p(‘para 1’) # <p>para 1</p>
    - h1 = createTag(‘h1’)
    - h1(‘header 1’) # <h1>header 1</h1>
* Decorator
  + way to modify the function result without changing the existing function code
  + to create a decorate a closure is required
  + prefix @ with the closure function to turn the closure into decorator
  + rules
    - outer function must accept a function alias (reference)
    - inner function must call the function (which is received as an input to outer function)
    - outer function must return reference of inner function (inner function name)
  + syntax:
    - def <outer>(<function>):
      * def <inner>(\*args, \*\*kwargs):
        + <function>(\*args, \*\*kwargs)
      * def <inner>
    - @<decorator>
    - def function1:
      * pass
  + e.g.
    - def log(func):
      * def inner(\*args, \*\*kwargs):
        + print(‘inside inner’)
        + print(func, args, kwargs)
        + func(\*args, \*\*kwargs)
      * return inner
    - @log
    - def add(p1, p2):
      * print(p1 + p2)
    - add(10, 20)
    - output:
      * inside inner
      * function add, (10, 20), {}
      * 30
* object oriented programming
  + terms
    - class
      * template to create an object
      * blueprint to create an object
      * instructions to create collection of attributes
      * memory will NOT be allocated for a class
      * types
        + empty class
        + concrete class

class having its body

* + - * + standalone class

class without parent class

* + - attribute
      * placeholder to hold a value/address
      * types
        + which stores a single value
        + which refers to a memory location where the method body gets loaded
    - reference
      * variable which stores address of an object
      * e.g.
        + p1 = Person()
        + where

p1: reference

* + - object
      * collection of attributes (methods are also referred as attributes internally)
      * also called as an instance of a class
      * memory will be allocated
      * to create object
        + <reference> = <class name>()
      * e.g.
        + p1 = Person()
        + where

Person() will create an object

* + - method
      * function declared inside a class
      * every method must declare the first parameter as self
      * method must be called on an object
      * e.g.
        + p1.printInfo()
* root class
  + a class considered as a parent/super/base class for all the classes in python
  + every python class is derived from root class directly or indirectly
  + in python 3.x, **object** is a root class
  + responsibility of root class
    - to provide default implementation (code) for the methods like \_\_str\_\_ etc
  + e.g.
    - # the Person class gets derived from object implicitly (automatically)
    - class Person:
      * pass
    - print(Person.\_\_bases\_\_) # (object)
    - # ths Car class is derived from object explicitly (manually)
    - class Car(object):
      * pass
    - print(Car.\_\_bases\_\_) # (object)
* type of methods
  + initializer
    - used to initialize the object
    - gets called automatically
    - gets called for every object for every object
    - can not be controlled
    - to add initializer
      * def \_\_init\_\_(self):
        + pass
  + setter
    - used to set a value of an attribute
    - syntax:
      * def <name>(self, <value>):
        + <body>
    - e.g.
      * def setPrice(self, price):
        + setattr(self, ‘price’, price)
  + getter
    - used to get/return a value of an attribute
    - e.g.
      * def getPrice(self):
        + return getattr(self, ‘price’)
  + facilator
    - which provides a facility
    - e.g.
      * def canVote(self):
        + if getattr(self, ‘age’) >= 18:

print(‘Yes’)

* + - * + else:

print(‘NO’)

* + deinitializer
    - used to de-initialize the object
    - gets called automatically when the object is getting deleted from memory
    - gets called for every object for every object
    - to create de-initializer
      * def \_\_del\_\_(self):
        + pass
    - to delete an object use del keyword
    - e.g.
      * class Test:
        + def \_\_init\_\_(self):

pass

* + - * + def \_\_del\_\_(self):

pass

* + - * t1 = Test() # \_\_init\_\_ will be called
      * del t1 # \_\_del\_\_ will be called
* Method overriding
  + can be done only in case of inheritance
  + child class implements a method with same name as that of the parent class
  + e.g.
    - class Person:
      * der printInfo(self):
        + print(‘inside Person’)
    - class Player:
      * der printInfo(self):
        + print(‘inside Player’)
    - p1 = Person()
    - p1.printInfo() # inside Person
    - p2 = Player()
    - p2.printInfo() # inside Player
    - note: here Player class is overriding printInfo method
* access specifiers
  + public:
    - the attribute can be accessed/modified outside the class
    - do not use \_ or \_\_ prefix
  + private:
    - the attribute can be accessed/modified only inside the class
    - the attribute can NOT be accessed/modified outside the class
    - use \_\_ prefix for an attribute to make it as private
    - convention
      * when a class provides a private member one must not use the member directly
  + protected:
    - the attribute can be accessed within the same and its child class(es)
    - to add protected member use \_ prefix
* operator overloading
  + giving more feature/meaning to existing operators
  + types
    - logical operators

|  |  |  |
| --- | --- | --- |
| **operator** | **method** | **example** |
| p1 < p2 | \_\_lt\_\_(self, other) | p1.\_\_lt\_\_(p2) |
| p1 <= p2 | \_\_le\_\_(self, other) | p1.\_\_le\_\_(p2) |
| p1 == p2 | \_\_eq\_\_(self, other) | p1.\_\_eq\_\_(p2) |
| p1 != p2 | \_\_ne\_\_(self, o) | p1.\_\_ne\_\_(p2) |
| p1 > p2 | \_\_gt\_\_(self, o) | p1.\_\_gt\_\_(p2) |
| p1 >= 2 | \_\_ge\_\_(self, power) | p1.\_\_ge\_\_(2) |

* + - mathematical operators

|  |  |  |
| --- | --- | --- |
| **operator** | **method** | **example** |
| p1 + p2 | \_\_add\_\_(self, other) | p1.\_\_add\_\_(p2) |
| p1 – p2 | \_\_sub\_\_(self, other) | p1.\_\_sub\_\_(p2) |
| p1 \* p2 | \_\_mul\_\_(self, other) | p1.\_\_mul\_\_(p2) |
| p1 / p2 | \_\_truediv\_\_(self, o) | p1.\_\_truediv\_\_(p2) |
| p1 // p2 | \_\_floordiv\_\_(self, o) | p1.\_\_floordiv\_\_(p2) |
| p1 \*\* 2 | \_\_pow\_\_(self, power) | p1.\_\_pow\_\_(2) |
| p1 % p2 | \_\_mod\_\_(self, other) | p1.\_\_mod\_\_(p2) |

* Code Reusability
  + inheritance
    - is also called as is-a relationship
    - Rule
      * the child class has to initialize the parent class object within it
      * e.g.
        + class Vehicle:

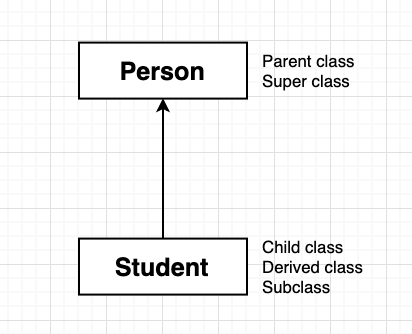
def \_\_init\_\_(self):

pass

* + - * + class Car(Vehicle):

def \_\_init\_\_(self):

Vehicle.\_\_init\_\_(self)

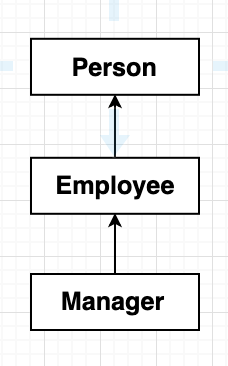
* + - e.g.
      * Bike is-a Vehicle
      * Crow is-a Bird
      * Student is-a Person
    - types
      * single inheritance
        + involves one parent and one child class
        + e.g.

class Person:

pass

class Student(Person):

pass

* + - * multi-level inheritance
        + the inheritance contains a chain of classes
        + types

direct inheritance

Employee is derived from Person directly

indirect inheritance

Manager is derived from Employee directly and from Person indirectly

* + - * + e.g.

class Person:

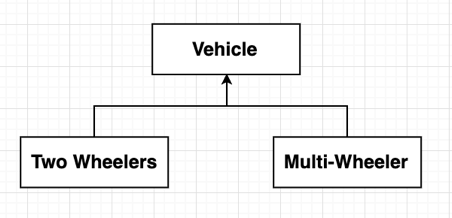
pass

class Employee(Person):

pass

class Manager(Employee):

pass

* + - * hierarchical inheritance
        + one parent class is used to create multiple child classes
        + e.g.

class Vehicle:

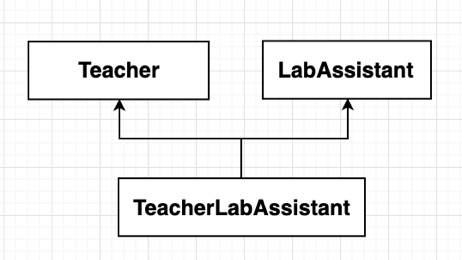
pass

class TwoWheelers(Vehicle):

pass

class MultiWheelers(Vehicle):

pass

* + - * multiple inheritance
        + one child class gets derived from multiple parent classes
        + e.g.

class Teacher:

pass

class LabAssistant:

pass

class TeacherLabAssistant:

pass

* + - * hybrid inheritance
        + combination of at least any two inheritance types
  + aggregation
    - is also called as has-a relationship
    - e.g.
      * Person has-a Car
      * Player has-a SportKit
      * Person has-a Address
    - e.g.
      * def Address:
        + pass
      * def Person:
        + def \_\_init\_\_(self, address):

self.address = address

* + - * a1 = Address()
      * p1 = Person(a1)
* exception/error handling
  + exception
    - a run-time/dynamic event/condition that may crash the application
    - e.g. ZeroDivisionError will be raised in case of division by 0
    - to raise/throw an exception, use raise keyword
    - e.g. raise ZeroDivisionError()
  + custom/user defined exception
    - a class derived from Exception can act as a custom exception
    - e.g.
      * class InvalidAgeException(Exception):
        + pass
      * raise InvalidAgeException()
  + handling an exception
    - avoiding the application crash by handling the exception with except block
  + blocks used in exception handling
    - try
      * block which contains statement(s) which may raise at least one exception
      * if exception occurs, the control leaves the try block and goes to one of the except blocks
    - except
      * used to handle an exception
      * types
        + specific

except block used to except only one type of except

* + - * + generic

used to handle any type of exception

if present, it must be the last except block

* + - else
      * contains statement(s) which need to be executed in case of no exception
    - finally
      * contains the statement(s) which need to be executed in case of exception or no exception
  + e.g.
    - def function1(p1, p2):
      * try:
        + result = p1 / p2
        + print(‘result: {}’.format(result))
        + raise IOError()
      * except ZeroDivisionError: # specific except block
        + print(‘ZeroDivisionError occurred’)
      * except:
        + print(‘generic except block’) # generic except block
      * else:
        + print(‘will executed in case of no exception’)
      * finally:
        + print(‘will be executed irrespective of exception raised’)
* modules
* packages
* third party packages
  + numpy
  + pandas
  + matplotlib
* sklearnkit