* Python
→ Easy to understand
→ Opensource
→ High level language - user friendly
→ Portable - multiple OS support
→ dynamically typed - type of variable is not declared during creation
→ interpreted language
→ created by Guido Van Rossum in 1989 launched in 1991

* Jupyter Notebook
→ open-source web application
→ allows us to create and share documents that contains live codes, equations, visualizations and narrative text.
→ increased productivity, improved communication, enhanced collaboration

NOTE Introspection - using question mark (?) before or after a variable will display some general info

* First Python Program

```
In [1]: print ("Hello World")
Out [1]: Hello World

In [2]: print ('Hello', 1, 4.5, True, sep='/')
Out [2]: Hello/1/4.5/True

In [3]: print ('Hello', end='-')
        print ('World')
Out [3]: Hello-World
```

* Data Types

Integers

print(8) Out: 8

range: (1e309)

Decimal / Float

print(8.5) Out[]: 8.5

range : (1.7e309)

Boolean

In[]: print(True) Out[]: True

In[]: print(False) Out[]: False

Complex

In[]: a = 5+6j

a.real OP[]: 5

a.imag OP[]: 6

String

In[]: print('Hello World')

Out[]: Hello World

List

In[]: print([1,2,3,4,5])

Out[]: [1,2,3,4,5]

Tuple

In[]: print((1,2,3,4))

Out[]: (1,2,3,4)

Set

In[]: print({1,2,3,3,4,5})

OP[]: {1,2,3,4,5}

Dictionary

```
In[]: print({'name':'dilkhush', 'gender':'male'})
Out[]: {'name':'dilkhush', 'gender':'male'}
```

None

```
In[]: b = None
Out[]:
```

NOTE type()- type() function tell about data type of argument

* Variables

→ Variable - name given to memory location

→ Keywords - Reserved words

→ Identifiers - class/function/variable/object name.

NOTE - Python supports dynamic typing & dynamic binding

Dynamic typing - data type of variable is not declared during creation & determined during runtime.

Ex- a = 9

Dynamic binding - variable can refer to a different type of object simply by assignment.

Ex-

```
In[1]: b = 5
In[2]: type(b)
Out[2]: int
In[3]: b = "foo"
Out[3]:
In[4]: type(b)
Out[4]: str
```

* Comments

```
# This is single line comment

"""This is
   a multiline
   comment
"""

'''This is
   also a multiline
   comment
'''
```

* Pip & ~~Not~~ Module

→ Pip is a package manager for python

→ Module is a file containing code written by programmes, which can be imported and used in our programs.

→ These are two types of modules -

1) Built-in - already available in python
   Ex- OS, time, abc
2) External - we have to install them externally.
   Ex- NumPy, tensorflow, Pandas

* Operators

1. Arithmetic +, -, *, /, //, %
2. Assignment =, +=, -=
3. Comparison ==, >, <, >=, <=, !, !=
4. Logical AND, OR, NOT
5. Membership in, not in

* Mutable & Immutable objects
→ Mutable - Object or values that they contain can be modified.
→ List, Dictionary, Numpy array, Set
→ Ex.

```
In[1]: a_list = ["foo", 2, [4,5]]
In[2]: a_list[2] = (3,4)
In[3]: a_list
Out[3]: ['foo', 2, (3,4)]
```

→ Immutable - objects or values that they contain can not be modified
→ String, Tuple
→ Ex

```
In[1]: a_tuple = (3, 5, (4,5))
In[2]: a_tuple[1] = "four"
TypeError
```

* Typecasting
→ used to convert a datatype to another suitable datatype.
→ These are two types -
1. Explicit typecasting - Programmer have to convert by using int(), float(), str(), list(), etc.

```
In[1]: a= "1"
In[2]: b= "2"
In[3]: int(a) + int(b)
Out[3]: 3
```

2. Implicit typecasting - Interpreter automatically converts the datatypes into required datatype

```
In[1]: 1.9 + 2
Out[1]: 3.9
```

* Taking User Input

```
In[1]: name = input()
In[2]: print("Your name is", name)
Out[2]: Your name is dilkhush
```

* Strings

→ Anything in between single or double quotes is a string. Now also between triple-single quotes is also a doc-string.

→
```
In[1] : name = "dilkhush"
In[2] : print("Your name is", name)
Out[2]: Your name is dilkhush
In[3] : name[0]
Out[3]: 'D'
In[4] : name[7]
Out[4]: 'h'
```

→
```
In[1] : name = 'dilkhush'
In[2]: print(len(name))
Out[2]: 8
In[3]: name[0:5]
Out[3]: dilkh
In[4] : name[0:9]
Out[4] : dilkhush
In[5]: name[0:-3]
Out[5] : dilkh
In[6]: name[-4:-2]
Out[6]: hu
```

* String Methods

1. str.upper()

creates a new string with uppercase

```
In[1]: a = 'string'
In[2]: a.upper()
Out[2]: STRING
```

2. str.lower()

creates a new string with lowercase

```
In[3]: a.lower()
Out[3]: string
```

3. str.swapcase()

interchange character casing

```
In[1]: b = 'hELLo, How aRE yOu?'
In[2]: b.swapcase()
Out[2]: 'HEllO, hOW ARe YoU?'
```

4. str.capitalize()

first char → uppercase, remaining → lowercase

```
In[1]: c = 'dilkhush Singh'
In[2]: c.capitalize()
Out[2]: 'Dilkhush singh'
```

5. str.rstrip()

removes any trailing (repeating) characters

```
In[1]: d = 'Heya!!a!'
In[2]: d.rstrip('a!')
Out[2]: Hey
```

6. str.replace()

replaces all occurence of a string with other

```
In[1]: a = 'Hello!!!Hello!!'
In[2]: a.replace('Hello', 'Hey')
Out[2]: 'Hey!!!Hey!!'
```

7. str.title()

captialize first letter of each word in string

```
In[1]: title1 = 'My name is dilkhush'
In[2]: title1.title()
Out[2]: 'My Name Is Dilkhush'
```

8. str.center()

aligns the string to the center as per parameter

```
In[1]: b = 'ABCD'
In[2]: b.center(50)
Out[2]: '                ABCD                '
```

9. str.count()

returns occurency of given value in a string

```
In[1]: c = 'ABCADAFG'
In[2]: c.count('A')
Out[2]: 3
```

10. str.find()

returns first index of sub string in string.
if not found then returns -1

```
In[1]: d = "Hello everyone"
In[2]: d.find('lo')
Out[2]: 3
```

11. str.endswith()

checks if string ends with given value.

```
In[1]: e = "Hello everyone"
In[2]: e.endswith('one')
Out[2]: True
In[3]: e.endswith('lo')
Out[3]: False
```

NOTE: str.startswith() works same like this.

12. str.isalpha()

returns True if string only consist of A-Z or a-z

```
In[1]: alpha = 'afsABC'
In[2]: alpha.isalpha()
Out[2]: True
In[3]: not-alpha = 'afs ABC'
In[4]: not-alpha.isalpha()
Out[4]: False
```

13. str.isalnum()

returns True if entire string consists A-Z or a-z or 0-9

14. str.islower()

returns True if all characters in lowercase

15. str.istitle()

returns True if first letter of each word is capitalized.

→ Split/Join

```
In[1]: 'hi my name is dilkhush'.split()
Out[1]: ['hi', 'my', 'name', 'is', 'dilkhush']
In[2]: ' '.join(['hi', 'my', 'name', 'is', 'dilkhush'])
Out[2]: 'hi my name is dilkhush'
```

* Control flow

→ if, elif, and else

Ex.-
```
x = -5
if x<0:
    print('It's negative')
elif x==0:
    print('Equal to zero')
elif 0<x<5:
    print('Positive but smaller than 5')
else:
    print('Positive and larger than or equal to 5')
```

→ for loops
Iterating over a collection (like a list or tuple)
Syntax
```
for value in collection:
    # do something with value
```

Ex.-
```
sequence = [1, 2, None, 4, None, 6]
total = 0
for val in sequence:
    if val is None:
        continue
    total += val
```

Ex.-
```
name = 'dilkhush'
for i in name:
    print(i)
```

Ex.-

```
users = {'Hans': 'active', 'Eleonore': 'inactive', 'sam': 'active'}
for user, status in users.copy().items():
    if status == 'inactive':
        del users[user]
active_users = {}
for user, status in users.items():
    if status == 'active':
        active_users[user] = status
print(f'Active users are {active_users}')
```

O/P- Active users are {'Hans': 'active', 'Sam': 'active'}

→ range

generates a sequence of evenly spaced integers.

```
In [1]: list(range(5))
Out[1]: [0, 1, 2, 3, 4]

In [2]: list(range(0, 20, 2))
Out[2]: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

In [3]: list(range(5, 0, -1))
Out[3]: [5, 4, 3, 2, 1]
```

NOTE- range produces integers up to but not including the endpoint.

* Built-In Data Structure

* Tuple

→ fixed length, immutable sequence, ordered

```
In[1]: tup = (4,5,6)    # tuple creation
In[2]: tup
Out[2]: (4,5,6)

In[3]: tup = 4,5,6
In[4]: tup
Out[4]: (4,5,6)

In[5]: tup = tuple('string')
In[6]: tup
Out[6]: ('s','t','r','i','n','g')    # indexing
In[7]: tup[0]
Out[7]: 's'

In[8]: nested_tup = (4,5,6),(7,8) #nested-tuple
In[9]: nested_tup
Out[9]: ((4,5,6),(7,8))
In[10]: nested_tup[0]
Out[10]: (4,5,6)
```

→ If an object inside a tuple is mutable, then you can modify it in place

```
In[1]: tup = tuple(['foo',[1,2],True])
In[2]: tup[1].append(3)
In[3]: tup
Out[3]: ('foo',[1,2,3],True)
```

→ we can concatenate tuples using + operator

```
In[1]: (4, None, 'foo') + (6.0) + ('bar',)
Out[1]: (4, None, 'foo', 6.0, 'bar')
```

→ multiplication is also possible

```
In[1]: ('foo', 'bar') * 4
Out[1]: ('foo', 'bar', 'foo', 'bar', 'foo', 'bar', 'foo', 'bar')
```

IMP unpacking tuple

```
In[1]: tup = (4,5,6)
Out[1]
In[2]: a,b,c = tup
In[3]: b
Out[3]: 5
```

IMP *rest function

```
In[1]: values = 1,2,3,4,5
In[2]: a,b,*rest = values
In[3]: a
Out[3]: 1
In[4]: b
Out[4]: 2
In[5]: rest
Out[5]: [3,4,5]
```

NOTE. # empty tuple created by parantheses

```
In[1]: empty = ()
In[2]: type(empty)
Out[2]: <class 'tuple'>
```

NOTE.

```
In[3]: tup('foo')
In[4]: type(tup)
Out[4]: <class 'str'>
```

→ Tuple methods

```
In[1]: a = 1,2,4,3,7,5
In[2]: a.count(2)
Out[2]: 1

In[3]: a.index(3)
Out[3]: 3
```

* List

→ variable length, mutable sequence, ordered

```
In[1]: list1 = []   # Empty list
In[2]: list2 = [1,2,3,4,5]
In[3]: list2
Out[3]: [1,2,3,4,5]

In[4]: L = [[[1,2],[3,4]],[[5,6],[7,8]]]
In[5]: L[0][0][1]
Out[5]: 2

In[6]: L = [1,2,3,4,5,6]
In[7]: L[-2:-5:-1]
Out[7]: [5,4,3]
```

→ Adding and removing elements

list.append(x) -

```
In[1]: a_list = ['foo','bar','baz']
In[2]: a_list.append('dwarf')
In[3]: a_list
Out[3]: ['foo','bar','baz','dwarf']
```

list.insert(index, val)

```
In[4]: a_list.insert(2, True)
In[5]: a_list
Out[5]: ['foo', 'bar', True, 'baz', 'qwasf']
```

list.pop(index)

```
In[6]: a_list.pop(4)  # By default last index
In[7]: a_list
Out[7]: ['foo', 'bar', True, 'baz']
```

list.remove(val)

```
In[8]: a_list.remove(True)
In[9]: a_list
Out[9]: ['foo', 'bar', 'baz']
```

→ Checking elements in list

```
In[10]: 'foo' in a_list
Out[10]: True
In[11]: 'foo' not in a_list
Out[11]: False
```

NOTE. Checking whether a list contains a value is a lot slower than doing so with dictionaries and sets, as Python makes linear scan across the values of the list, whereas it can check the others (based on hash tables) in constant time.

→ concatenating and combining lists

```
In[1]: L1 = [1,2,3,4]
In[2]: L2 = [5,6,7,8]
In[3]: L1+L2
Out[3]: [1,2,3,4,5,6,7,8]

In[4]: L1.extend([4,None,'foo'])
In[5]: L1
Out[5]: [1,2,3,4,4,None,'foo']
```

NOTE- extend() is faster than concatenating by +

→ sorting

```
In[1]: L = [2,4,7,1,5,9,6]
In[2]: L.sort(reverse=True) #sorting descending
In[3]: L
Out[3]: [9,7,6,5,4,2,1]
In[4]: L.sort(reverse=False) # by default
In[5]: L
Out[5]: [1,2,4,5,6,7,9]

In[6]: sorted(L, reverse=True)
In[7]: L
Out[7]: [9,7,6,5,4,2,1]
In[8]: sorted(L, reverse=False)
In[9]: L
Out[9]: [1,2,4,5,6,7,9]

In[10]: L = ['saw','small','he','foxes','six']
In[11]: L.sort(key=len)
In[12]: L
Out[12]: ['He','saw','six','small','foxes']
```

→ List functions

```
In[1]: L = [2,1,5,7,0]
In[2]: len(L)
Out[2]: 5
In[3]: min(L)
Out[3]: 0
In[4]: max(L)
Out[4]: 7
```

list.count(val)

```
In[5]: L.count(0)
Out[5]: 1
```

list.index(val)

```
In[6]: L.index(5)
Out[6]: 2
```

list.reverse()

```
In[7]: L.reverse()
Out[7]: [0,7,5,1,2]
```

list.copy()

```
In[8]: M = L.copy()
In[9]: M
Out[9]: [2,1,5,7,0]
In[10]: M.append(9)
In[11]: M
Out[11]: [2,1,5,7,0,9]
In[12]: L
Out[12]: [2,1,5,7,0]
```

* Dictionary
→ collection of key values.
→ mutable, ordered, key's can't be duplicated
→ Creation

```
In[1]: d={}
In[2]: type(d)
Out[2]: <class 'dict'>

In[3]: d={'a': 'some value', 'b':[1,2,3,4]}
In[4]: d
Out[4]: {'a':'some value', 'b':[1,2,3,4]}
```

→ Accessing items

```
In[5]: d['a']
Out[5]: 'some value'
In[6]: d.get('a')
Out[6]: 'some value'
```

NOTE. dict.get('key') method doesn't throw any error if requested key is not present in dictionary.

```
In[7]: d.keys()
Out[7]: dict_keys(['a','b'])

In[8]: d.values()
Out[8]: dict_values(['some value',[1,2,3,4]])
```

→ ~~words classifies by alphabets~~

```
for key, value in d1.items():
    print(f'value corresponds to key
        {key} is {value}')
```

→ words classifies by alphabets

```
words = ['apple', 'bat', 'Bar', 'atom', 'book', 'Dog', 'don']
by_letter = {}
for word in words:
    letter = word[0].capitalize()
    if letter not in by_letter:
        by_letter[letter] = [word]
    else:
        by_letter[letter].append(word)
by_letter
```

O/P - {'A': ['apple', 'atom'], 'B': ['bat', 'Bar', 'book'], 'D': ['Dog', 'don']}

→ Dictionary methods

d1.update(d2)

```
In[1]: d1 = {1:1, 5:5, 7:7}
In[2]: d2: {4:4, 10:10, 12:12}
In[3]: d1.update(d2)
In[4]: d1
Out[4]: {1:1, 5:5, 7:7, 4:4, 10:10, 12:12}
In[5]: d2
Out[5]: {4:4, 10:10, 12:12}
```

d1.pop(key)

```
In[6]: d1.pop(10)
Out[6]: 10
In[7]: d1
Out[7]: {1:1, 5:5, 7:7, 4:4, 12:12}
```

dict.popitem()

```
In [8]: d1.popitem # no argu dict last item
In Out[8]: (10,12)
In [9]: d1
Out[9]: {1:1, 5:5, 7:7, 4:4}
```

del d1[key]

```
In[10]: del d1[5]
In[11]: d1
Out[11]: {1:1, 7:7, 4:4}
```

d1.clear()

```
In [12]: d2.clear()
In [13]: d2
Out [13]: {}
```

del d1

```
In [14]: del d2
In [15]: d2
NameError: name 'd2' is not defined
```

membership

```
In[16]: 7 not in d1
Out [16]: False
```

len

```
In[17]: len(d1)
Out[17]: 3
```

* Set
- unordered collection
- unique items
- mutable
- can't contain mutable types.

- Creation

```
In[1]: empty-set = set()
In[2]: empty-set
Out[2]: set()
In[3]: s1 = set([0,1,2,3])
In[4]: s1
Out[4]: {0,1,2,3}
In[5]: s2 = {1, 'hello', (1,2)}
In[6]: s2
Out[6]: {(1,2), 1, 'hello'}
```

- Accessing

```
In[7]: for s in s2:
           print(s)
Out[7]: (1,2)
        1
        'hello'
```

- Adding / Removing items

```
In[1]: s = {1,2,3,4}
In[2]: s.add(5)
In[3]: s
Out[3]: {1,2,3,4,5}

In[4]: s.update([6,7])
In[5]: s
Out[5]: {1,2,3,4,5,6,7}
```

```
In[6]: s.discard(5)
In[7]: s
Out[7]: {1,2,3,4,6,7}

In[8]: s.remove(6)
In[9]: s
Out[9]: {1,2,3,4,7}
```

NOTE- discard doesn't throw error if item is absent.

```
In[10]: s.pop()
Out[10]: 1
In[11]: s
Out[11]: {2,3,4,7}

In[12]: s.clear()
In[13]: s
Out[13]: set()

In[14]: del s
In[15]: s
NameError: name 's' is not defined
```

→ Set operations

a.union(b) - all of the unique elements in a and b

```
In[1]: a = {1,2,3,4,5}
In[2]: b = {3,4,5,6,7,8}
In[3]: a.union(b)
Out[3]: {1,2,3,4,5,6,7,8}
In[4]: a | b
Out[4]: {1,2,3,4,5,6,7,8}
```

```
In [5]: a.update(b)   # a |= b
In [6]: a
Out[6]: {1,2,3,4,5,6,7,8}
```

a.intersection(b) - all of the elements in both a and b

```
In [7]: a.intersection(b)
Out[7]: {3,4,5}
In [8]: a & b
Out[8]: {3,4,5}
In [9]: a.intersection_update(b)  # a &= b
In [10]: a
Out[10]: {3,4,5}
```

a.difference(b) - the elements in a that are not in b

```
In [1]: a = {1,2,3,4,5}
In [2]: b = {3,4,5,6,7,8}
In [3]: a.difference(b)
Out[3]: {1,2}
In [4]: a - b
Out[4]: {1,2}
In [5]: b.difference(a)
Out[5]: {6,7,8}
In [6]: b - a
Out[6]: {6,7,8}

In [7]: a.difference_update(b)  # a -= b
In [8]: a
Out[8]: {1,2}
```

a.symmetric-difference(b) - all of the elements in either a or b but not both

```
In[1]: a = {1,2,3,4,5}
In[2]: b = {3,4,5,6,7,8}
In[3]: a.symmetric_difference(b)
Out[3]: {1,2,6,7,8}
In[4]: a ^ b
In[9]: a.sy  Out[9]: {1,2,6,7,8}
```

```
In[5]: a.symmetric_difference_update(b)
In[6]: a
Out[6]: {1,2,6,7,8}
In[7]: a ^= b
In[8]: a
Out[8]: {1,2,6,7,8}
```

a.issubset(b) - True if the elements of a are contained in b

```
In[1]: s1 = {1,2,3,4}
In[2]: s2 = {2,3}
In[3]: s1.issubset(s2)
Out[3]: False
In[4]: s2.issubset(s1)
Out[4]: True
In[5]: s1 <= s2
Out[5]: False
In[6]: s2 <= s1
Out[6]: True
```

a.issuperset(b) - True if the elements of b are all contained in a

```
In[7]: a.issuperset(b)
Out[7]: True
In[8]: b.issuperset(a)
Out[8]: False
In[9]: a >= b
Out[9]: True
In[10]: b >= a
Out[10]: False
```

a.isdisjoint(b) - True if a and b have no elements in common

```
In[1]: s1 = {1,3,5,7}
In[2]: s2 = {2,4,6,8}
In[3]: s1.isdisjoint(s2)
Out[3]: True
```

Copying a set

```
In[4]: s3 = s1.copy()
In[5]: s3
Out[5]: {1,3,5,7}
```

Membership

```
In[6]: 5 in s1
Out[6]: True
In[7]: 11 not in s2
Out[7]: True
```

* Frozenset
→ Immutable version of set
→ all read operations are applicable
→ write operations doesn't work

```
In [1]: fs1 = frozenset([1,2,3])
In [2]: fs2 = frozenset([3,4,5])
In [3]: fs1 | fs2
Out[3]: frozenset({1,2,3,4,5})

In [4]: fs1.add([2,3,8])
AttributeError: 'frozenset' object has
                no attribute 'add'

In [5]: fs1.cleas()
AttributeError: 'frozenset' object has
                no attribute 'cleas'
```

*

* Built-In Sequence Function

→ enumerate

returns a sequence of (i, value) tuple

```
In [1]: l = [1,2,3]
In [2]: for i, val in enumerate(l):
            print (f'value at {i} index is {val}')
Out[2]: value at 0 index is 1
        value at 1 index is 2
        value at 2 index is 3
```

→ Sorted

returns new sorted list from the elements of any sequence, optional arg reversed = True/False

```
In [3]: sorted (l, reversed = True)
Out[3]: [3, 2, 1]
In [4]: sorted ('hello')
Out[4]: ['e', 'h', 'l', 'l', 'o']
```

→ zip

pairs up the elements of a number of lists, tuples, or other sequences to create a list of tuples.

```
In[5]: l2 = ['A', 'B', 'C', 'D']
In[6]: zipped = zip (l, l2)
In [7]: list (zipped)
Out[7]: [(1,'A'), (2,'B'), (3,'C')]
```

NOTE. zip() can take arbitrary number of sequences, and the no of elements it produces is determined by the shortest sequences.

→ reversed

reversed iterates over the elements of a sequence in reverse order.

```
In[1]: tup = 1,2,4,6,8,10
In[2]: list(reversed(tup))
Out[2]: [10,8,6,4,2,1]
```

* List, Set, and Dictionary Comprehensions

→ List Comprehension

form a new list by filtering the elements of a collection, transforming the elements passing the filter into one concise expression.

→ List of squares from 1 to 10

```
L = []
for i in range(1,11):
    L.append(i**2)
print(L)
```

O/P - [1,4,9,16,25,36,49,64,81,100]

OR

```
L=[i**2 for i in range(1,11)]
print(L)
```

O/P- [1,4,9,16,25,36,49,64,81,100]

→ List of all numbers divisible by 5 in range of 0 to 51

```
In[1]: l2=[i for i in range(1,51) if i%5==0]
In[2]: l2
Out[2]: [5,10,15,20,25,30,35,40,45,50]
```

newlist = [expression for item in iterable if condition == True]

Date: / /

Nested List Comprehension

```
tup = ((1,2,3),(4,5,6))
flat_tup = []
for x in tup:
    for y in x:
        flat_tup.append(x)
print(flat_tup)
```

O/P - [1, 2, 3, 4, 5, 6]

OR

```
tup_flat = [y for x in tup for y in x]
print(tup_flat)
```

O/P - [1, 2, 3, 4, 5, 6]

→ Set Comprehension

same like list comprehension here we use curly braces instead of square brackets

```
In [1]: {i**2 for i in range(1,11) if i%2==0}
Out [2]: {4, 16, 36, 64, 100}
```

→ Dictionary Comprehension

```
In [1]: {i: i**2 for i in range(1,11) if i%2==0}
Out [1]: {2:4, 4:16, 6:36, 8:64, 10:100}

In [1]: days = ['sun', 'mon', 'tue', 'wed', 'thus', 'fri', 'sat']
In [2]: temp = [30.5, 32.6, 28.7, 30.1, 29.7, 31.8, 28.6]
In [3]: {i:j for (i,j) in zip(days, temp)}
Out [3]: {'sun': 30.5, 'mon': 32.6, 'tue': 28.7,
          'wed': 30.1, 'thus': 29.7, 'fri': 31.8,
          'sat': 28.6}
```

* Functions (code modularity, readibility, reusability)

→ Two types:

1. Built in functions
2. Uses defined functions

```
In[1]: def is_even(num):
           """
           This function returns if a given number is even or odd
           input - any valid integer
           output - even/odd
           created on - 21st Mar 2024
           """
           if type(num) == int:
               if num%2 == 0:
                   return 'even'
               else:
                   return 'odd'
           else:
               return 'invalid input'

In[2]: is_even(3)
Out[2]: 'odd'

In[3]: is_even('hello')
Out[3]: 'invalid input'

In[4]: print(is_even.__doc__)
Out[4]: This function returns if a given number is even or odd
        input - any valid integer
        output - even/odd
        created on - 21st Mar 2024
```

NOTE- If there is no return type in function then it returns None.

Date: / /

→ Parameter - variables that are defined or declared when the function is created.

→ Argument - values that are passed into the function during function calling.

→ Types of Arguments

1. Default arguments
2. Positional arguments
3. keyword arguments

```
In[1]: def power(a=1,b=1):
           return a**b
```

```
#default
In[2]: power()
Out[2]: 1
```

```
#positional
In[3]: power(2,3)
Out[3]: 8
```

```
#keyword
In[4]: power(b=3,a=3)
Out[4]: 27
```

→ *args and **kwargs

special python keywords that are used to pass the variable length of arguments to a function.

NOTE- if a function uses normal, *args and **kwargs then order of arguments must be normal > *args > **kwargs

→ *args

- *args allows us to pass a variable length or variable numbes of non-keyword arguments to a function.

```
In[1]: def multiplies(*args):
           product = 1
           for i in args:
               product *= i
           print(args)
           return product
```

```
In[2]: multiplies(1,2,3,4,5,6,7,8,9,10,12)
Out[2]: (1,2,3,4,5,6,7,8,9,10,12) ## treated as tuple
        43545600
```

→ **kwargs

- **kwargs allows us to pass any numbes of keyword arguments.

- keyword arguments mean that they contain a key-value pais like python dictionary.

```
In[1]: def display(**kwargs):
           for(key,value) in kwargs.items():
               print(key,'->',value)
```

```
In[2]: display(india='delhi','srilanka'='colombo','nepal'=
               'kathmandu')
Out[2]: india -> delhi
        srilanka -> colombo
        nepal -> kathmandu
```

Date: / /

→ Variable Scope

I/P- def g(y):
    print(x)
    print(x+1)
x=5
g(x)
print(x)

O/P- 5
6
5

I/P- def f(y):
    x = 1
    x += 1
    print(x)
x = 5
f(x)
print(x)

O/P- 2
5

I/P- def h(y):
    x += 1
x = 5
h(x)
print(x)

O/P- UnboundLocalError: Can't modify global variable from a function without using global keyword.

I/P-
```
def f(x):
    x = x+1
    print ('in f(x): x=', x)
    return x
x=3
z=f(x)
print ('in main program scope: z=', z)
print ('in main program scope: x=', x)
```

O/P-
```
in f(x): x = 4
in main program scope: z=4
in main program scope: x=3
```

* Nested function

I/P-
```
def f():
    def g():
        print ('inside function g')
    g()
    print('inside function f')
f()
```

O/P
```
inside function g
inside function f
```

I/P-
```
def g(x):
    def h():
        x = 'abc'
    x += 1
    print ('in g(x): x=', x)
    h()
    return x
x=3
z=g(x)
```

O/P- in g(x): x = 4

I/P-
```
def g(x):
    def h(x):
        x = x+1
        print("in h(x): x = ", x)
    x = x+1
    print("in g(x): x = ", x)
    h(x)
    return x
x = 3
z = g(x)
print("in main program scope: x = ", x)
print("in main program scope: z = ", z)
```

O/P-
```
in g(x): x = 4
in h(x): x = 5
in main program scope: x = 3
in main program scope: z = 4
```

* FUNCTIONS ARE 1ST CLASS CITIZENS

→ supports functions as arguments to other functions, returning them as values from other functions, assigning them to variables or storing them in data structures.

```
In[1]: def square(num):
           return num**2
#type
In[2]: type(square)
Out[2]: function
#id
In[3]: id(square)
Out[3]: 2803315256640
```

```
#reassign
In[4]: x = square
In[5]: x(3)
Out[5]: 9
In[6]: id(x) == id(square)
Out[6]: True
#deleting
In[7]: del square
In[8]: square(3)
NameError: name 'square' is not defined.
#storing
In[9]: L = [1,2,3,4, square]
In[10]: L[-1](4)
Out[10]: 16
#function is immutable
In[11]: s = {square}  #sets only contain immutable
In[12]: s
Out[12]: {<function --main--.square(num)>}
#returning a function
In[13]: def f():
            def x(a,b):
                return a+b
            return x
In[14]: val = f()(3,4)
Out[14]: 7
#function as arguments
In[15]: def func-a():
            print('inside func-a')
        def func-b(z):
            print('inside func-b')
            return z()
        print(func-b(func-a))
Out[15]: inside func-b
         inside func-a
```

* Lambda Function
- small anonymous function
- can take any number of arguments, but can only have one expression.

```
In[1]: a = lambda a,b,c: a+b+c
In[2]: a(4,3,8)
Out[2]: 10
```

→ Difference b/w lambda and normal function

1. No name
2. lambda has no return value (infact returns a function)
3. Lambda function is written in one line.
4. not reusable

NOTE- Lambda functions are specially used higher order functions.

→ Lambda function to check whether string contains 'a'

```
In[1]: a = lambda s: 'a' in s
In[2]: a('hello')
Out[2]: False
In[3]: a('abcd')
Out[3]: True
```

→ lambda function to check whether the num is even/odd

```
In[1]: n = lambda x: 'even' if x%2==0
                     else 'odd'
In[2]: n(5)
Out[2]: odd
In[3]: n(8)
Out[3]: even
```

* Higher Order functions
- functions which accepts another function as argument or returns another function

```
In[1]: def transform(f,L):
           output=[]
           for i in L:
               output.append(f(i))
           print(output)

In[2]: L=[1,2,3,4,5]
In[3]: transform(lambda x:x**2,L)
Out[3]: [1,4,9,16,25]
```

* Map
- map function applies a function to each element in a sequence
- returns map object that can transformed into sequence

```
In[1]: l=[1,2,3,4,5,6]
In[2]: sq = list(map(lambda x:x*x,l))
In[3]: sq
Out[3]: [1,4,9,16,25,36]
```

* Filter
- filter function filters elements of a sequence based on given predicate

```
In[1]: l=[5,6,1,7,8,4]
In[2]: newl = list(filter(lambda x:x>5,L))
In[3]: newl
Out[3]: [6,7,8]
```

```
In[1]: fruits = ['apple', 'guava', 'mango', 'cherry']
In[2]: list(filter(lambda x: x.startswith('a'), fruits))
Out[2]: ['apple']
```

* Reduce
- applies a function to sequence and returns a single value.
- It is part of 'functools' module

```
In[1]: from functools import reduce
In[2]: n = [1,2,3,4,5,6]
In[3]: sum = reduce(lambda x,y: x+y, n)
In[4]: sum
Out[4]: 21
```

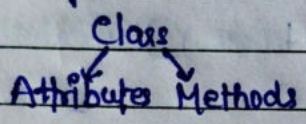
* OOP - Object Oriented Programming

6 Principles of OOP-

1. Class
2. Object
3. Polymorphism
4. Encapsulation
5. Inheritence
6. Abstraction

→ Class

- Class is a blueprint.

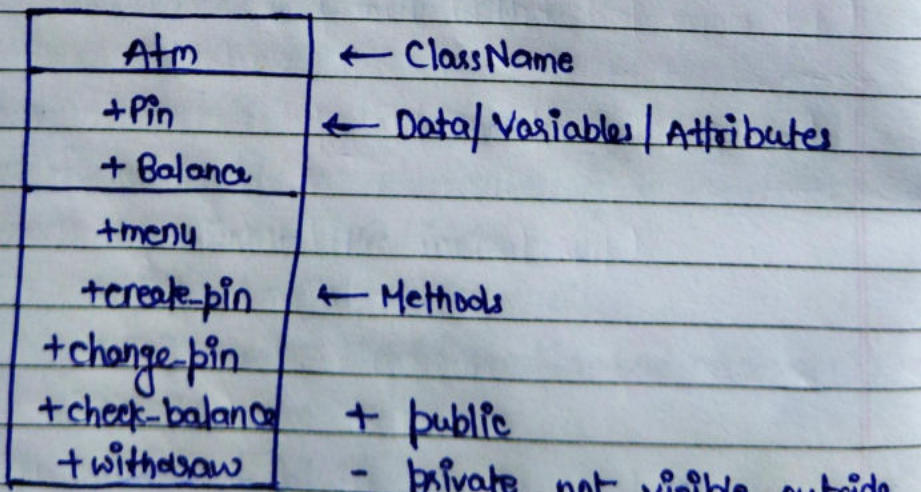


→ Object

- Instance of a class.

NOTE- Class name should be in Pascal Case (Every word starts with capital letter) E.g. HelloWorld, MyClass

→ Class Diagram



```
class Atm:
    # constructor
    def __init__(self):
        self.pin = ''
        self.balance = 0
        self.menu()
    def menu(self):
        user_input = input ("""
            Hi, How can I help you?
            1. Press 1 to Create Pin
            2. Press 2 to Change Pin
            3. Press 3 to Check Balance
            4. Press 4 to Withdraw
            5. Anything else to exit.
        """)
        if user_input == '1':
            self.create_pin()
        elif user_input == '2':
            self.change_pin()
        elif user_input == '3':
            self.check_balance()
        elif user_input == '4':
            self.withdraw()
        else:
            exit()
    def create_pin(self):
        user_pin = input ('Enter your pin')
        self.pin = user_pin
        user_balance = input('Enter your balance')
        self.balance = user_balance
        print('Pin created successfully')
        self.menu()
```

```
    def change_pin(self):
        old_pin = input('Enter current pin')
        if old_pin == self.pin:
            new_pin = input('Enter new pin')
            self.pin = new_pin
            print('Pin changed successfully')
        else:
            print('Pin mismatched')
        self.menu()
    def check_balance(self):
        user_pin = input('Enter your pin')
        if user_pin == self.pin:
            print('Your balance is', self.balance)
        else:
            print('Pin mismatched')
        self.menu()
    def withdraw(self):
        user_pin = input('Enter your pin')
        if user_pin == self.pin:
            amount = int(input('Enter amount'))
            if amount <= self.balance:
                self.balance -= amount
                print('Withdrawal successful')
            else:
                print('Insufficient balance')
        else:
            print('Pin mismatched')
        self.menu()

obj1 = Atm()
```

O/P Hi, How can I help you? 1. Press 1 to Create Pin 2. Press 2 to Change Pin 3. Press 3 to Check Balance 4. Press 4 to Withdraw 5. Anything else to exit

→ Methods v/s Functions

Method - function implemented within a class.

Function - Independently created function.

E.g.
```
In[1]: L=[1,2,3,4]
In[2]: len(L)  # function
In[3]: L.append(5)  # Method
```

→ Magic Methods (Dunder methods)

special methods

e.g. --init--

→ Constructor (--init--)

constructor contains configuration related things.

invoked automatically when an object is created.

Two types -

1. Parameterized - accepts arguments along with self.
2. Parameterless - don't accept any arguments from object.

→ How objects access attributes

```
class Person:
    def __init__(self, name, country):
        self.name = name
        self.country = country
    def greet(self):
        if self.country == 'india':
            print('Namaste', self.name)
        else:
            print('Hello', self.name)
# creating object
p = Person('dilkhush', 'india')
```

```
# accessing attributes
p.name
O/P 'dilkhush'

# accessing methods
p.greet()
O/P - Namaste dilkhush

# tying to access non-existent attributes
p.gender
O/P- AttributeError : 'Person' object has no attribute 'gender'

# attribute creation from outside the class
p.gender = 'male'
p.gender
O/P - 'male'
```

* Reference Variables

- Reference variables hold the object.
- We can create objects without reference variable as well
- An object can have multiple reference variables.
- Assigning a new reference variable to an existing object does not create a new object.

```
class Person:
    def __init__(self):
        self.name = 'dilkhush'
        self.gender = 'male'
p = Person()
q = p
```

multiple ref

```
print(id(p))
```

o/p - 27688799906944

```
print(id(q))
```

o/p - 27688799906944

changing attribute value with 2nd reference variable

```
print(p.name)
print(q.name)
q.name = 'ankit'
print(p.name)
print(q.name)
```

o/p - dilkhush
dilkhush
ankit
ankit

* Encapsulation

→ Instance vars - variables that have different values for different objects.

```
class Student:
    def __init__(self, name, id):
        self.id = id      # id is private variable
        self.name = name
        self.__id = id

    # Getter
    def get_id(self):
        return self.__id
    # Setter
    def set_id(self, new_id):
        self.__id = new_id

s = student('dilkhush', 8)
s.get_id()
o/p - 8
s.set_id(7)

s1 = Student('ankit', 4)
s2 = Student('rahul', 15)

L = [s, s1, s2]
for i in L:
    print(i.name, i.get_id())
```

o/p - dilkhush 7
ankit 4
rahul 15

* Static Variable

- instance var - unique for every object of the class
- static var - same for all the objects of the class

```
class Student:
    __ids = 1      # ids is static variable
    def __init__(self, name):
        self.name = name
        self.__id = Student.__ids
        Student.__ids += 1

    #utility method
    @staticmethod
    def get_ids():
        return Student.__ids
```

→ static attributes are created at class level.
→ static attributes are accessed using ClassName
→ static attributes are object independent, we can use them without creating instance of the class.
→ The value stored in static attribute is shared between all instances of the class in which the static attribute is defined.

* Class Relationships

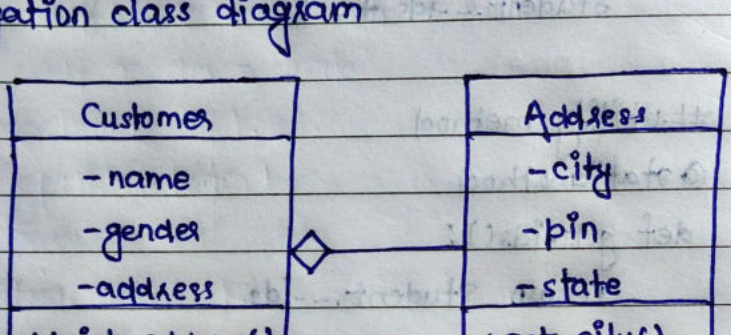
1. Aggregation
2. Inheritence

→ Aggregation

- has-a relation

e.g Customer has a Address.
Student has a Result.

- Aggregation class diagram



- class Customer:

```
def __init__(self, name, gender, address):
    self.name = name
    self.gender = gender
    self.address = address
def print_address(self):
    print(self.address.get_city(), self.address.pin,
          self.address.state)
def edit_profile(self, new_name, new_city, new_pin, new_state):
    self.name = name
    self.address.edit_address(new_city, new_pin, new_state
```

```
class Address :
    def __init__(self, city, pin, state):
        self.__city = city
        self.pin = pin
        self.state = state
    def get_city(self):
        return self.__city
    def edit_address(self, new_city, new_pin, new_state):
        self.__city = new_city
        self.pin = new_pin
        self.state = new_state

add1 = Address('mandsaur', 458001, 'MP')
cust1 = Customer('dilkhush', 'male', add1)
cust1.print_address()
```

O/P- mandsaur 458001 MP

```
cust1.edit_profile('rahul', 'indore', 450002, 'MP')
cust1.print_address()
```

O/P - indore 450002 MP

NOTE- You can't access private variables in aggregation.

→ Inheritence

child class can access public attributes and methods of the parent class.

What gets inherited ?

- Constructor
- Non private attributes
- Non private methods

- Inheritence class diagram


```
#Parent Class
class User:
    def __init__(self):
        self.name = 'dilkhush'
        self.gender = 'male'
    def login(self):
        print('Login')
#Child Class
class Student(User):
    def __init__(self):
        self.rollno = 100
    def enroll(self):
        print('enrolled into the course')
#Creating object
u = User()
s = Student()
print(s.name)
```

O/P - AttributeError: 'Student' object has no attribute 'name'

NOTE - Explanation - since child also have constructor then priority of child class constructor is high and it gets executed. same with methods having same name in child & parent class.

```
print(u.name)
```

O/P - dilkhush

```
s.login()
```

O/P - login

* Super keyword
- way to access parent's methods only constructor including
- can't access data attributes.

```
class Phone:
    def __init__(self, price, brand, camera):
        print('Inside phone constructor')
        self.__price = price
        self.brand = brand
        self.camera = camera
    def buy(self):
        print('Buying a Phone')

class SmartPhone(Phone):
    def buy(self):
        print('Buying a smartphone')
        super().buy()

s = SmartPhone(74000, 'APPLE', 12)
s.buy()
```

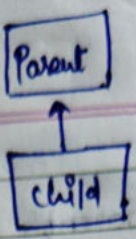
O/P - Inside phone constructor
Buying a smartphone
Buying a Phone

NOTE- super keyword can not be used outside the class.

→ Types of Inheritence
1. Single Inheritence
2. Multilevel Inheritence
3. Hierarchical Inheritence
4. Multiple Inheritance
5. Hybrid Inheritence

Date: /

1. Single Inheritence

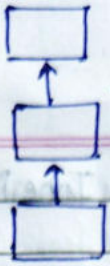



```
class Phone:
    def __init__(self, price, brand, camera):
        print('Inside phone constructor')
        self.__price = price
        self.brand = brand
        self.camera = camera
    def buy(self):
        print('Buying a Phone')

class SmastPhone(Phone):
    pass

SmastPhone(1000, 'Nokia', '2mp').buy()
```

O/P - Inside phone constructor
Buying a Phone

Date: / /

2. Multilevel Inheritence



```
class Product:
    def review(self):
        print('Product customer review')

class Phone(Product):
    def __init__(self, price, brand, camera):
        print('Inside phone constructor')
        self.__price = price
        self.brand = brand
        self.camera = camera
    def buy(self):
        print('Buying a Phone')

class SmartPhone(Phone):
    pass

s = SmartPhone(70000, 'Apple', 12)
```

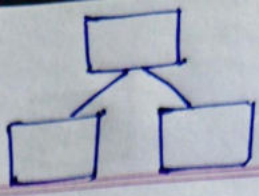
O/P - Inside phone constructor

```
s.buy()
```

O/P - Buying a Phone

```
s.review()
```

O/P - Product customer review



Date: /

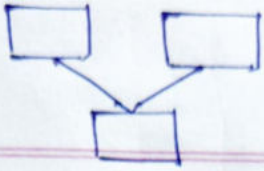
3. Hierarchical Inheritence

```
class Phone :
    def __init__(self, price, brand, camera):
        print('Inside Phone constructor')
        self.__price = price
        self.brand = brand
        self.camera = camera
    def buy(self):
        print('Buying a phone')

class SmartPhone(Phone):
    pass

class FeaturePhone(Phone):
    pass

SmartPhone(70000, 'Apple', 12).buy()
```

O/P - Inside Phone constructor
Buying a phone

```
FeaturePhone(1000, 'Nokia', 2).buy()
```

O/P - Inside Phone constructor
Buying a phone



Date: / /

4. Multiple Inheritence

```
class Product:
    def review(self):
        print ('Customer review')

class Phone:
    def __init__(self, price, brand, camera):
        print('Inside phone constructor')
        self.__price = price
        self.brand = brand
        self.camera = camera
    def buy (self):
        print ('Buying a phone')
    def review(self):
        print ('Phone review')

class SmartPhone(Product, Phone):
    pass

s = SmartPhone (70000, 'Apple', 12)
O/P - Inside phone constructor

s.buy()
O/P - Buying a phone

s.review()    # review of Product class
O/P - Customer review
```

NOTE- If multiple parent classes have methods with same name then order of inheritence matter

* Polymorphism
- ability of objects to respond differently to the same method call.

1. Method Overriding
2. Method Overloading
3. Operator Overloading

1. Method Overriding
- if parent class and child class both have methods of same name then child class method will precedence.

```
class Phone:
    def __init__(self, price, brand):
        self.__price = price
        self.brand = brand

class SmartPhone(Phone):
    def __init__(self, price, brand, camera):
        self.__price = price
        self.brand = brand
        self.camera = camera

p = SmartPhone(12000, 'Redmi', 50)
```

NOTE- Always constructor of child class gets executed.

2. Method Overloading
- multiple methods having same name but output is different based on input.
- used for clean code.

```
class Shape:
    def area(self, a, b=0):
        if b==0:
            return 3.14*a*a
        else:
            return a*b

s= Shape()
print(s.area(2))
```

O/P - 12.56

```
print(s.area(3,4))
```

O/P - 12

3. Operator Overloading
- Same operator showing diff behaviour based on input.

```
'hello'+'world'
```

O/P - 'helloworld'

```
4+5
```

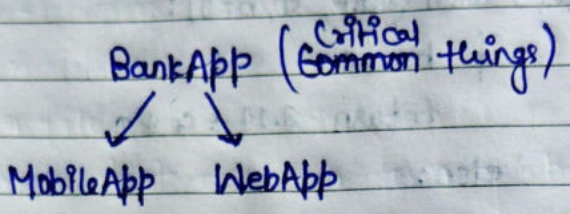
O/P - 9

```
[1,2,3]+[4,5]
```

O/P - [1,2,3,4,5]

* Abstraction
- allows us to focus on the essential details of a program by hiding unnecessary complexity.
- improved code reusability
- enhanced modularity
- increased flexibility

Eg




```
from abc import ABC, abstractmethod
class BankApp:
    def database(self):
        print('connection to Database')

    @abstractmethod
    def security(self):
        pass

    @abstractmethod
    def display(self):
        pass

class MobileApp(BankApp):
    def mobile_login(self):
        print('login into mobile')
    def security(self):
        print('mobile security')
    def display(self):
        print('display')
```

mob = MobileApp()
mob.security()
O/P - mobile security

obj = BankApp()
O/P - TypeError : Can't instantiate abstract class BankApp

NOTE- You can't create object of abstract class.

NOTE Can't ~~inst~~ create object of ~~eta~~child class until it contains all the abstract methods.

* File Handling

Date: / /

Types of data used for I/O :

1. Text - '1234' as a sequence of unicode chars
2. Binary - 1234 as a sequence of bytes of its binary equivalent

Types of files -

1. Text files - Text
2. Binary files - Image, music, video, exe, etc

→ Writing to a file

can only write text data in form of string datatype

```
f = open('sample.txt','w')
f.write('Hello World')  ## writing single line
f.close()
```

NOTE- If file is not present then it will get created

```
L = ['hello\n', 'hi\n', 'how are you\n', "I'm fine"]
f = open('samples.txt', 'w')
f.writelines(L)  ## writing multiple lines
f.close()
```

NOTE - If we try to write existing file then new content replace old one. If we want to merge then append mode is used.

```
f = open('sample.txt', 'a')
f.write('\nHow are you?')
f.close()
```

→ Reading from files

```
f = open('sample.txt', 'r')
s = f.read()    # Read whole file
print(s)
f.close()
```

O/P - Hello World
How are you?

```
# reading upto n chars
f = open('sample.txt', 'r')
s = f.read(10)
print(s)
f.close()
```

O/P - Hello Worl

```
# readline() - read line by line
f = open('sample1.txt', 'r')
print(f.readline(), end='')
print(f.readline(), end='')
f.close()
```

O/P - hello
hi

```
# reading entire file using readline
f = open('sample.txt', 'r')
while True:
    data = f.readline()
    if data == '':
        break
    else:
        print(data, end='')
f.close()
```

O/P - Hello World
How are you?

* Using Context Managers (With).

- It's good idea to close a file after usage as it will free up the resources.
- If we dont close it, garbage collector would close it.
- with keyword closes the file as soon as the usage is over.

```
with open('sample.txt', 'a') as f:
    f.write('created by dilkhush singh')
```

* Seek & Tell

→ seek()

seek() function allows us to move the cussent position within a file to a specific location.

```
with open('sample.txt','w') as f:
    f.write('Hello')
    f.seek(0)
    f.write('X')
    p= f.read()
print(p)
```

o/p - Xello

→ tell()

tell() function returns the cussent position

```
with open('sample.txt','r') as f:
    f.read()
    f.tell()
    print(f.tell())
    f.seek(0)
    print(f.tell())
```

o/p - 25
0

Working with binary files

```
with open('picture.png','rb') as f:
    with open('picture-copy.png','wb') as g:
        g.write(f.read())
```

* Serialization & Deserialization

→ Serialization - process of converting python data type to JSON format

→ Deserialization - process of converting JSON to python data types.

```
# serialization using JSON module
# list
import json
L = [1,2,3,4]
with open('demo.json','w') as f:
    json.dump(L,f)
```

```
# desesialization
import json
with open('demo.json','r') as f:
    d=json.load(f)
    print(d)
    print(type(d))
```

O/P - [1,2,3,4]
<class 'list'>

```
# dictionary
d = {'name': 'dilkhush', 'age': 20, 'gender': 'male'}
with open('demo.json', 'w') as f:
    json.dump(d, f, indent=4)

with open('demo.json', 'r') as f:
    d = json.load(f)
    print(d)
    print(type(d))
```

O/P- {'name': 'dilkhush', 'age': 20, 'gender': 'male'}
<class 'dict'>

serializing & deserializing custom objects

- we can't directly serialize and deserialize custom objects however it can be done using ~~method~~ function

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

p = Person('dilkhush', 20, 'male')

def show_object(person):
    if isinstance(person, Person):
        return {'name': person.name, 'age': person.age,
                'gender': person.gender}
with open('demo.json', 'w') as f:
    json.dump(p, f, default = show_object, indent=4)
```

```
with open('demo.json','r') as f:
    q = json.load(f)
    print(q)
    print(type(q))
```

O/P - {'name':'dilkhush', 'age':20, 'gender':'male'}
<class 'dict'>

NOTE. After deserialization object is converted to dictionary and it loses all its properties. Hence we overcome it by pickling.

* Pickling

Pickling - python object hierarchy → byte stream
Unpickling - reverse of pickling

```
import pickle
with open('demo.pkl','wb') as f:
    pickle.dump(p,f)

with open('demo.pkl','rb') as f:
    q = pickle.load(f)
    print(q.name)
```

O/P - dilkhush

→ Pickle v/s JSON
- Pickle lets the user to store data in binary format.
- JSON lets the user to store data in human-readable text format.

Date: / /

* Exception Handling

There are two stages where errors may happen in prog

1. During compilation → Syntax Error
2. During execution → Exception

* Syntax Error

- raised by interpreter / compiler
- can be solved by rectifying the program

E.g • Leaving symbols like colon, bracket

```
In[1]: print 'hello world'
SyntaxError: Missing parantheses...
```

```
In[2]: a = 5
       if a => 3
           print ('hello')
SyntaxError: invalid syntax
```

• Misspelling a keyword

```
In[3]: iff a==3:
           print(a)
SyntaxError: invalid syntax
```

• Incorrect indentation

```
In[4]: if a==5:
       print(a)
IndentationError: expected an indented block
```

• empty if / else / loops / class / function

```
In[5]: for i in range(5):
SyntaxError: incomplete input
```

IndexError

The IndexError is thrown when trying to access an item at invalid index.

```
In[1]: L = [1,2,3,4,5]
In[2]: L[50]
IndexError: list index out of range
```

ModuleNotFoundError

thrown when a module could not be found.

```
In[1]: import mathi
ModuleNotFoundError: No module named 'mathi'
```

KeyError

when a key is not found

```
In[1]: d = {'name': 'dilkhush'}
In[2]: d['age']
KeyError: 'age'
```

TypeError

when an operation is applied to unsupported type.

```
In[1]: 1 + 'a'
TypeError: unsupported operand type(s) for +: 'int'
    and 'str'
```

ValueError

when a function's argument is of inappropriate type.

```
In[1]: int('a')
ValueError: invalid literal for int() with base 10: 'a'
```

NameError

when a variable is not defined.

Attribute Error

when a object doesn't have requested attribute.

* Exceptions
- raised by python runtime
- solved by logical programming.

E.g. • Error in connection to external source (DB | file | etc)

~~In: with open('sample.txt', 'w') as f:~~
~~f.write~~
~~p=f.read()~~
~~print(p)~~

In[1]: with open('file.txt', 'r') as f:
    p = f.read()
    print(p)

O/P- FileNotFoundError: No such file or directory

• Divide by 0 → logical errors

In[2]: print(5/0)

O/P- ZeroDivisionError: division by zero

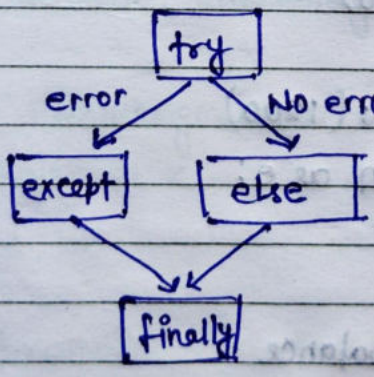
• Memory overflow
- when size of file exceeds available memory space.

→ Why is it important to handle exceptions?

1. User experience → error messages can ruin UX
2. Security → along with error message it also shows the code line, file name and many more critical informations.

In [1]:
```
try:
    with open('sample.txt','r') as f:
        print(f.read())
    print(m)
    print(5/0)
except FileNotFoundError:
    print('File not found.')
except NameError:
    print('Variable not defined')
except ZeroDivisionError:
    print('Can not divide by 0')
except Exception as e:
    print(e)
else:
    print('No error till now')
finally:
    print("I will execute, i don't care about errors")
```

O/P - Hello World
Variable not defined
I will execute, i don't care about errors



else: if there is no error at all then control go to else block directly from try block.

finally: No matter what happens finally will always executed.

→ raise Exception

manually raising exceptions also optionally we can pass values. ↓

In[1]: raise NameError ('trial purpose')
O/P- NameError: trial purpose.

```
class Bank:
    def __init__(self, balance):
        self.balance = balance
    def withdraw(self, amount):
        if type(amount) != int:
            raise Exception('amount should be numeric')
        if amount<0:
            raise Exception('amount can not be -ve')
        if self.balance <amount:
            raise Exception('Insufficient balance')
        self.balance -= amount

obj = Bank(1000)
try:
    obj.withdraw(1200)
except Exception as e:
    print(e)
else:
    print(obj.balance
```

O/P- Insufficient balance

We can create our own class for exceptions

→ usecase - application based task

```
class Security(Exception):
    def __init__(self, msg):
        print(msg)
    def logout(self):
        print('functionality to logout')
class Google:
    def __init__(self, name, email, passw, device):
        self.name = name
        self.email = email
        self.passw = passw
        self.device = device
    def login(self, email, passw, device):
        if device != self.device:
            raise Security('Suspected device detected')
        if email == self.email and passw == self.passw:
            print('login successful')
        else:
            print('login error')
obj = Google('dilkhush', 'dilkhush@gmail.com', 1234, 'android')
try:
    obj.login('dilkhush@gmail.com', 1234, 'windows')
except Security as e:
    e.logout()
else:
    print(obj.name)
finally:
    print('database connection closed')
```

O/P- Suspected device detected
functionality to logout
database connection closed

* Namespace

- A namespace is a space that holds names (identifiers)
- namespace are dictionary of identifiers (keys) & objects (values)

4 types -

1. Local namespace
2. Enclosing namespace
3. Global namespace
4. Builtin namespace

→ Scope

A scope is textual region of program where a namespace is directly accessible

→ LEGB Rule

The interpreter searches for an identifier from inside out, looking in local, enclosing, global and finally the builtin scope. If the interpreter doesn't find the name in any of these locations, then it raises a NameError excepti

```
#global var
a = 5
def temp():
    #local var
    b = 3
    print(a, b)
temp()
```

o/p 5, 3

NOTE - We can access global variable in local scope but can't access local variable outside its scope

C In:
```
a=5
def temp():
    a=3
    print(a)
temp()
print(a)
```
O/P- 3
5
According to LEGB rule priority is given to local var

NOTE- We can access but can't modify global variable from local scope directly. However we can modify or create by using 'global' keyword.

In:
```
a=5
def temp():
    a+=2
    print(a)
temp()
```
O/P- UnboundLocalError: cannot access local variable 'a' where~

In:
```
a=5
def temp():
    global a
    a+=2
    print(a)
    global b
    b=5
    print(b)
temp()
print(a,b)
```
O/P- 7
5
7, 5

built-in scope

In:
```
import builtins
print(dir(builtins))
```
O/P- ['ArithmeticError', ..., 'None', ..., 'max', ..., 'zip']

In:
```
L = [1,2,3]
print(max(L))
def max():
    print('hello')
print(max(L))
```
O/P- 3

TypeError: max() takes 0 positional arguments but 1 was give

Exp: - Interpreter works line by line, when first max(L) comes it searches L → G → E → B and max() is builtin function so it prints the maximum but in second time we created another max() functio and it takes no arguments hence producing error.

enclosing scope

- local scope inside another local scope

In:
```
def outer():
    a = 1
    def inner():
        nonlocal a      # nonlocal works same as global
        a += 1          #   for enclosing scope
        print('inner', a)
    inner()
    print('outer', a)
outer()
```
O/P- inner 2
outer 2

* Decorators

Date: / /

- function that receives another function as input and adds some functionality (decoration) to it and returns

Two types -

1. Built in decorators

@staticmethod, @abstractmethod, @property, @classmethod

2. User-defined decorators

```
def my-decorator(func):
    def wrapper():
        print('Welcome')
        func()
        print('Thanks for using function')
    return wrapper

@my-decorator
def hello():
    print('hello')
hello()
```

o/p - Welcome
hello
Thanks for using function

```
impost time
def times(func):
    def wrapper(*args):
        start = time.time()
        func(*args)
        print('time taken by', func.__name__,
              time.time() - start, 'secs')
    return wrapper

@times
def hello():
    print('hello')
    time.sleep(1)

@times
def square(num):
    print(num * num)
    time.sleep(1)

@times
def power(a,b):
    print(a**b)
    time.sleep(1)
hello()
square(25)
power(25,2)
```

O/P -

```
hello
time taken by hello 1.003621 secs
625
time taken by square 1.001020 secs
625
time taken by power 1.000532 secs
```

Date: / /

* Iteration

- Repetitive execution of same block of code over and over.

```
In: num = [1,2,3]
    for i in num:
        print(i)
```

```
O/P: 1
     2
     3
```

→ Iterators

- object that allows the programmer to traverse through a sequence of data without having to store the entire data in memory.
- every iterator has 'iter' and 'next' function

```
In: L = [x for x in range(10000000)
    import sys
    print(sys.getsizeof(L))
    x = range(10000000)
    print(sys.getsizeof(x))
```

```
O/P. 89095160
     48
```

→ Iterable

- Iterable is an object which ~~can be~~ one can iterate over.
- every iterable has 'iter' function

```
In: L = [1,2,3]
    type(iter(L))
```

```
O/P. list_iterator
```

NOTE

Every iterator is also iterable
Not all iterables are iterators

In: num = [1,2,3]

```
#fetch the iterators
iter_num = iter(num)
print(next(iter_num))
print(next(iter_num))
print(next(iter_num))
```

O/P- 1
2
3

In: `print(next(iter_num))`

O/P- StopIteration:

```
#Our own for loop
def for_loop(iterable):
    iterators = iter(iterable)
    while True:
        try:
            print(next(iterators))
        except StopIteration:
            break
a = [1,2,3]
for_loop(a)
```

O/P- 1
2
3

In: `b = range(2)`

O/P- 0
1
2

* Generators

Date: / /

Generators are a simple way to creating iterators.

```
# iterators without generators
class MyRange:                    # Iterable
    def __init__(self, start, end):
        self.start = start
        self.end = end
    def __iter__(self):
        return MyIterator(self)

class MyIterator:
    def __init__(self, iterable)
        self.iterable = iterable
    def __iter__(self):
        return self
    def __next__(self):
        if self.iterable.start >= self.iterable.end:
            raise StopIteration
        current = self.iterable.start
        self.iterable.start += 1
        return current
```

In:
```
a = MyRange(2, 10)
for i in a:
    print(i)
```

O/P.
```
2
3
4
5
6
7
8
9
```

simple generator

```
In: def gen_demo():
        yield 'first statement'
        yield 'second statement'
        yield 'third statement'

In: gen = gen_demo()
    for i in gen:
        print(i)
```

O/P- first statement
second statement
third statement

Generator Expression

```
In: gen = (i**2 for i in range(1,10))
    for i in gen:
        print(i, end=' ')
```

O/P- 1 4 9 16 25 36 49 64 81

Benefits

1. Ease of Implementation
2. Memory efficient
3. Representing infinite streams
4. Chaining generators

```
In: def fib_num(num):
        x, y = 0, 1
        for _ in range(num):
            x, y = y, x+y
            yield x
    print(sum(fib_num(10)))
```

O/P- 143