Broad Sum up of Python

- → Intro. to python
- → Python & OOPS
- → Python for datascience

Computer [Store & process]

↓

S/w & H/w (req.)

↓

Program (set of instruction in particular prg lang. for specific task)

↓

Product development lifecycle

Python [more features, usability, community (open source), frequent updates on lib., HLL, easier than C, C++, OOPS]

1 req. Analysis (client req.) → 2 plan the logic (Algorithm) → 3 Coding (in prg. lang.) → 4 HLL to LLL (by 'compiler / interpreter) → 5 Testing (a Syntax b semantic c logic) → 6 production based on feedback → back to 1

* Algorithm (step by step inst. for specific task) → Two main types → pseudocode [Read, print, loop, repeat]
  → flowchart (visual) 

* Compiler V/s Interpreter → 1 Store (entire exec. code) ? / 2 execute (entire / line by line) / 3 Bug find easy? / 4 faster?

* PRG. LANG. [PYTHON] I/P, process, O/P

↓

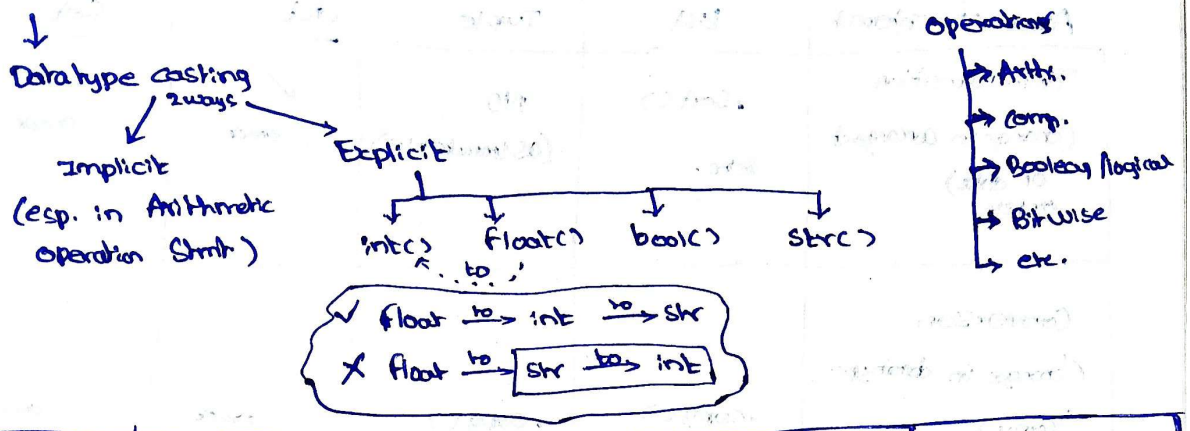
I/P → input( ) → runtime variable; default type is str.

O/P → print( ) → single/double/triple coats; f string (dynamic value change); end = xx (for multi print stmts); sep = xx (for single print stmt)

3 Datatypes ⇒ Datatype Conversion / casting

4 Variable (container)

Datatypes → str, numeric (int, float → complex), None, bool, Collection (it is object that can hold arbitrary objects) → list, dict, set, tuple → Mutable, Immutable



Area of Interest	List	Tuple	Dict	Set
Define	Ordered ('index access yes') collection of objects.	Ordered ('indexable' in it) Collection of objects [Immutable Tuple]	Key:value pair ↓ immutable (hashable) ↓ mutable	1 unordered (no indexing) collection of 2 distinct 3 hashable objects.
Mutability (hasability)	yes	NO* But objects which are mutable as elements of tuple are still mutable. *imp	yes ✓dict ✓dict.values() ✗dict.keys()	yes
Creation (change in size)	1 [ ] 2 list(...) 3 .to_list() 4 .append() 5 .extend()	1 tuple() 2 (x,)	1 {k1:v1, k2:v2, ...} 2 .setdefaults() 3 .update() 4 .fromkeys() 5 d[<keyx>] = <valx>	1 ( ) check 2 set( )
modification (no change in size)	1 l[<index>] = <val>	NO (as immutable)	1 d[<key>] = <value>	1 S.add(<val>)
Removal	1 del l[<index>] 2 del l	1 del t	1 del d 2 del d[<key>] 3 d.pop(<key>) 4 d.popitems(<key>)	1 S.remove() 2 S.pop() (; arbitary element removal.
Access (read specific data)	1 l[<index>]	1 t[<index>]	1 d[<key>] 2 d.items() 3 d.values() 4 d.keys()	NO (as no indexable)
Query (metadata)	.min(), .max(), .count(), etc	"	"	" check

Area of Interest	List	Tuple	dict	Set
Transformation (change in arrangement of data) order.	.sort() etc.	NO (as immutable)	✓ check	✓ check
Conversion (change in datatype & copy)	.copy()	.copy()	check	check
Set operation	X	X	X	.union() .intersection() .difference() .symmetric_diff_() with_update
Iterable	✓	✓	✓	X

* eval().

Python & OOPS

→ Focus on object & class than function & logic.

- object
  - A follows defined blueprint
  - B Actual data
- class
  - A Blueprint (define attribute & method)
  - B NO actual data

→ Attributes / properties / features / variables / data member

↳ define the state of the object of the class.

→ Method / member function

↳ function specific to that particular class.

→ Attribute types / classification

- on basis of instance specific / class specific
  - a Instance attri.
  - b Class attri.
- on basis of access modifier
  - a public
  - b private

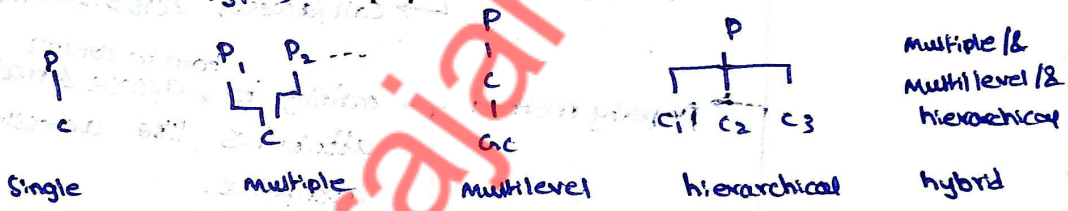
Method classification

- **Instance method**
  - first parameter is object itself
  - special types of it
    - works on esp. instance attri.
    - 1 Constructor
      - def. a `__init__()`
      - call. b `<classname>()`
    - 2 Destructor
      - a `__del__()`
      - b `del <object>`
- **Class method**
  - first parameter is class itself.
  - `@classmethod`
  - call: a `<obj>.<method>` b `<cls>.<method>`
  - works on esp. class attri.
- **Static method**
  - works on given parameter.
  - `@staticmethod`
  - works on logic bound to that class.
  - call/access: `<cls>.<method>`
- **Property method**
  - Mainly to control how to access & modify the instance attributes like accessing like attributes.
  - 3 main types
    - Getter → to derive the calculated value.
      - `@<prop.name>`
      - *access by `<obj>.<prop.name>`
    - Setter → validate / rule logic
      - `@<prop.name>.setter`
      - *access by = stmt.
    - deleter → free up space/resource
      - `@<prop.name>.deleter`
      - *access by `del <obj>.<prop.name>`
- **Magic method**
  - Mainly to change how the inBuilt method & operators behave based on object. (for predefined operations)
  - Constructor, destructor
    - `__init__()` `__del__()`
  - arithmetic op.
    - `__add__()` `__truediv__()` `__mod__()`
    - `__sub__()` `__floordiv__()` etc.
  - Comp. op.
    - `__eq__()` `__le__()`
    - `__lt__()` `__gt__()` etc.

OOPS Concept

1. Encapsulation
   - bind data + method (in class & obj.)
   - Mainly to restricted / controlled access of data member & methods.
   - ⇓
   - using access modifiers
     - public (can access outside the class directly)
     - private (need to / must to access by the member functions)
       - e.g. getter / setter.

2. Inheritance
   - class (child) derive methods & data members from the class (parent)
   - Mainly to reuse, extend, modify.
   - Types



Single | Multiple | Multilevel | hierarchical | hybrid (Multiple & Multilevel & hierarchical)

- Multiple → method conflict ⇓ resolve thru. Method resoultion order (MRO)
  - access:
    - <classname> <attri/method> (for all parent class)
    - super().<attri./method> (only to 1st parent class).
- hierarchical → Shared method & attri. among children.

3. polymorphism
   - Mainly to define custom behavior of member function (i.e. methods) and operators [inbuilt or user defined]
   - Specific to particular object.
   - Acheived thru.
     1. method overloading
     2. method overriding (Inheritance)
     3. Duck typing (dynamic typed python)

But method must be written / present in that class. <= reference object interchangable as python dynamic typed.

↳ 4 Abstraction

→ 1 Simplifies the complexity (✓what ; ✗how)

2 by mention method name (✓what)

3 but hide implementation details (how).

→ 1 from abc import ABC (Abstract class import it)

2 @abstractmethod

↳ implement the details of abstract method in the child class. => So, it just provide the template / blueprint (what) w/o how (details).

Functions

→ def <funname> (var1 : <dty1 / dty2 ...> , var2 : <dty1 / dty2, ...> , ...) → <dty1 / dty2 ...>

→ Variables (based on scope / visibility)

→ Global → to all.
→ Non local → outside of nested funct. is func.
→ local → in the func.

global sc > ?
local sc > to see them.

→ Function type

→ parameterized func. further → default keyword func.
→ Non parameterized func.

→ Argument type

→ default
→ positional
→ keyword
→ positional variable length arg. (*a)
→ keyword variable length arg. (**a)

* → after keyword arg. only
/ → Before positional arg only

return types.

Special features

- lambda function => nameless function. [Just execute & throw]

  lambda <args> : <expr.>

- Map function => maps each iterable with respective function's arguments

  map (func, [itr1, itr2, ...])

- Reduce => aggregated value of the iterable.

  reduce (func, [itr]) → only one iterable

- filter => filter outs the conditional satisfying elements of an iterable.

  filter (func, itr) → only one iterable

Numpy and pandas

* Number python
* Mathematical & Scientifical calculation on ndimensional array
* Homogenous datatype & shape.
* Contigous memory allocation
  (as homogenous datatype
  ↓
  same size
  ↓
  access element by simple formula from base address)

* creation
* access
  - index => fancy indexing → selective indexing
  - conditional subset/boolean filtering

* attributes → ndim, shape, size, dtype
* methods → concatenate, multiplication (matrix), stacking, transpose, split, array split, etc.

* panel data → tabular format
  ↓
  heterogeneous datatype
* For data analysis & manipulation
* Heterogeneous datatype & Homogenous shape
  (↳ with auto np.nan assign.
  ⇓ so
* contigous memory allocation in various blocks for each datatype.

* DataStructure in pandas
  - series → it is homogenous datatype
    - index, name, (+np array properties) + extra methods.
  - DataFrame → it can be heterogenous datatype.
    - index, columns, dtypes, (+np array prop.) + extra methods.

* attributes like ndim, shape, size, dtypes, etc.
* methods → concat, pivot_table, groupby, sort_values, merge, reset_index, drop, fillna, duplicated, drop_duplicates, etc.

Algorithm

- 1 Step wise 2 fesible (time) 3 Specific task 4 definiteness (unambiguous instructn) 5 finite (no. of steps).
- helps in 1 code refactoring, 2 reuse, etc.

Sorting → Quick sort (pivot); Bubble Sort (1st max in pass 1, ...)

- arrange in order (desc / asc)
- as function (not inplace, returns) => Sorted( )
  as method (inplace, not returns) => .sort( )

DataStructure

- to efficiently 1 organise 2 storage 3 manage data.
- mainly two types
  - 1 Linear (only one directional option in traversing) ; seq. ; fwd / bckd.
  - 2 Non linear (multiple directional options in traversing) ; hierarchical / random

e.g.

1 array → seq. ; continous storage ; homogenous datatype ; predefined (size in older lang.)

2 Stack → linear + LIFO
- operation → push, pop, etc

3 Queue → linear + FIFO

4 linked list
- Node: Data part | Next part
- linear ; may not be continous storage
- type – Singly, doubly, Circular
- add ; Delete

e.g.

Tree
- nonlinear
- parent (one node) can have multiple child (nodes)
- terms: leaf node, height of tree, depth of node, etc.
- Binary tree
  - @ most two nodes for a node.
  - types - complete, etc.
  - Sort, Search are fastener than array, list, Stack, etc.