Python Concepts Tour

Dachuri Chaitanya

M.Tech (IIT Roorkee), AI Expert

Trainer Introduction

- Post graduate from IIT Roorkee
- 20+ yrs Industry Experience
- Worked in TCS, HCL, TechM, Capgemini, etc…
- 10+ yrs Training Experience
- 60+ Batches, 4800+ students, 75000+ Weekly Tests, 9450+ Mock Interviews….

Course Specialties

- Session recording
- Course Material
- Highlight IMP
- Weekly Tests, Interviews
- 5 months
- Code notebook
- Assignments
- Anytime Questions
- Accessible Trainer
- 100% Job Assistance

DOs & DON'Ts

DOs

- Clear doubts immediately
- Spend min. 2-6 hrs per day
- Better to have own laptop
- Practice all assignments
- See absent class recordings ASAP

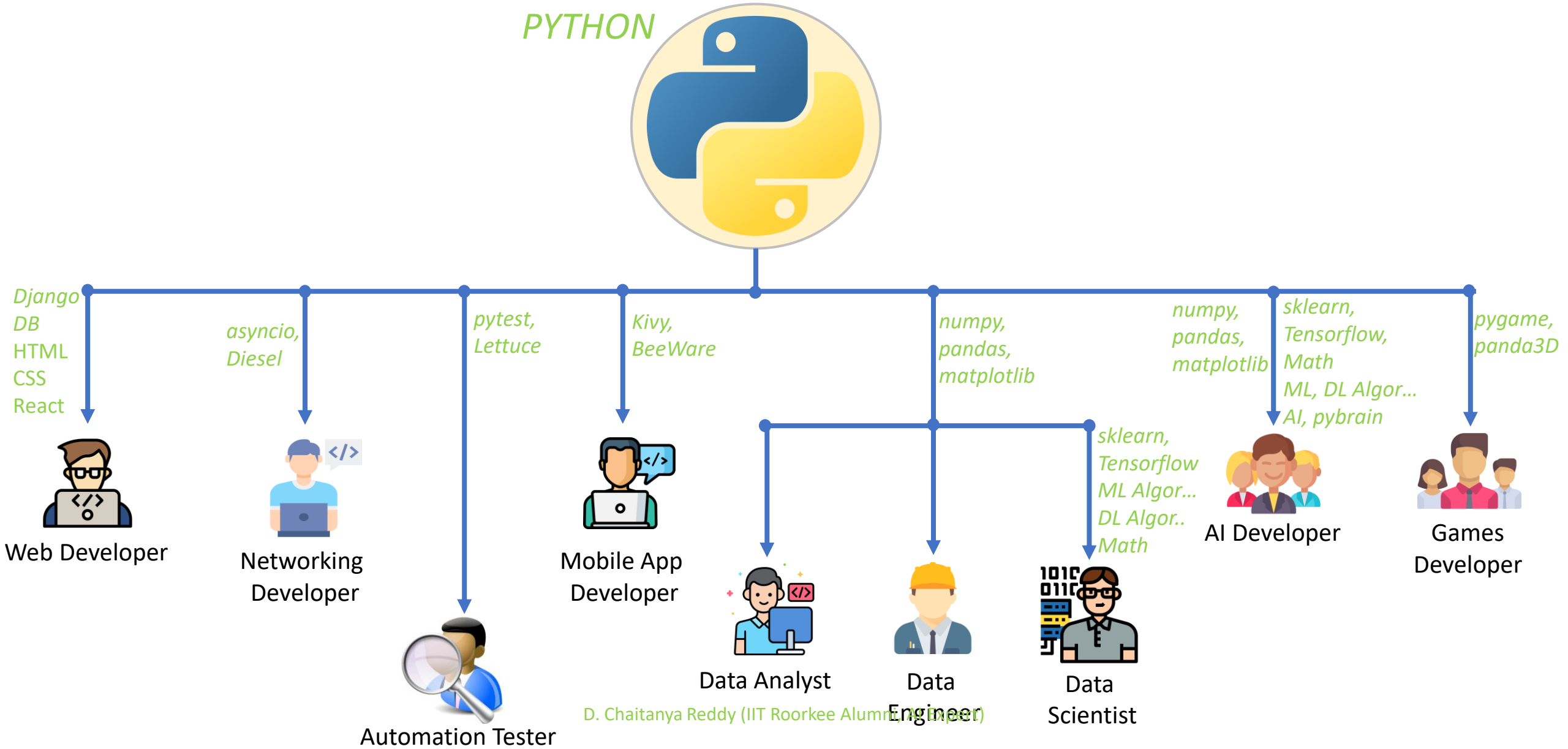
DON'Ts

- Do not absent to classes
- Do not take long leaves
- Do not be late to classes
- No need to write Notes in class
- Do not take food before class

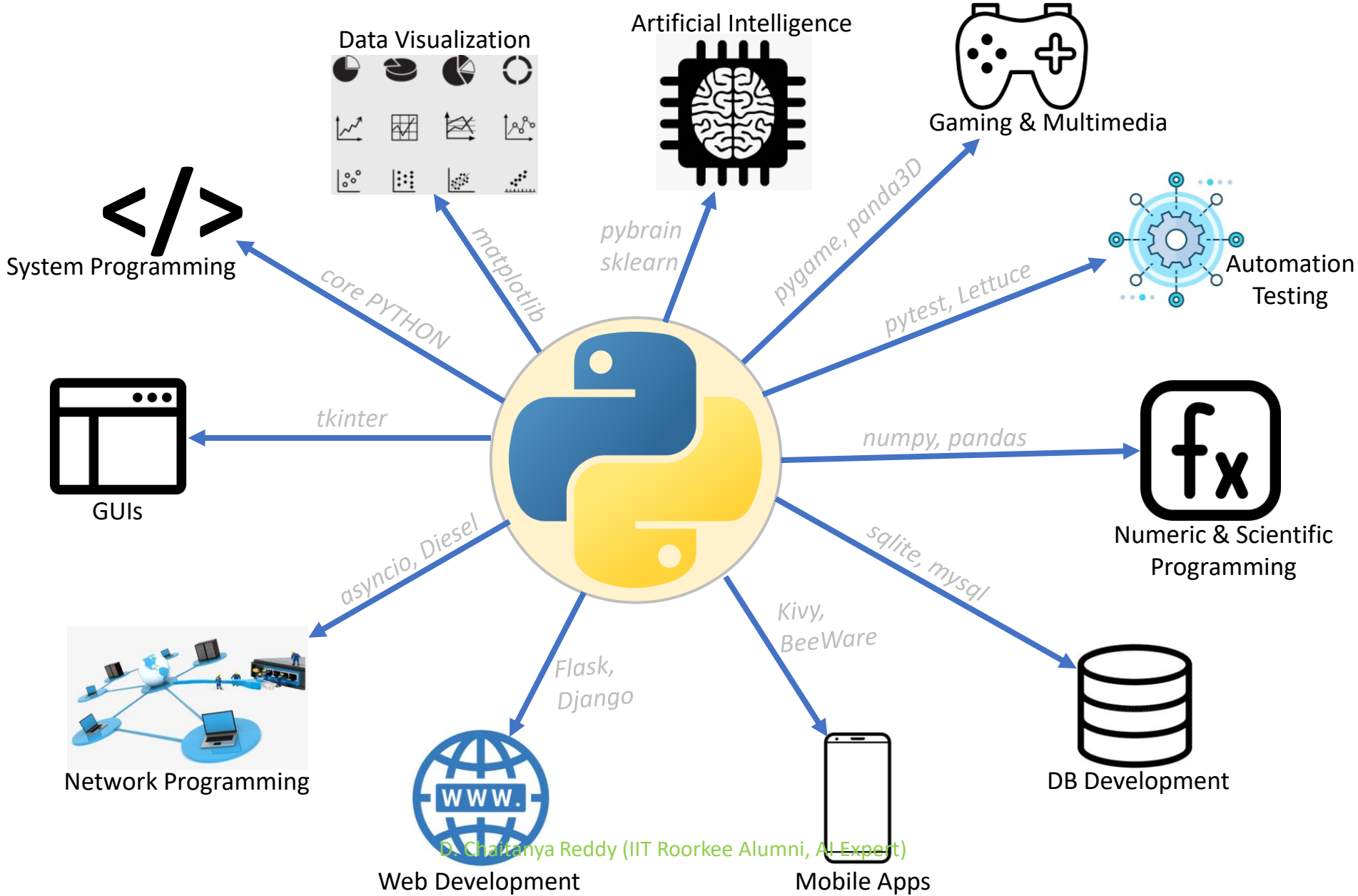
At the end of Course...

- Becomes equal to ~ 3 yrs exp. professional Python Full stack developer.
- Gains more than enough knowledge & hands on required for IT Job.
- Can easily crack interviews and get job in few days.
- Gets recommendation to jobs in all tied up start up IT companies.

Career options with PYTHON



What can do with **PYTHON** ?

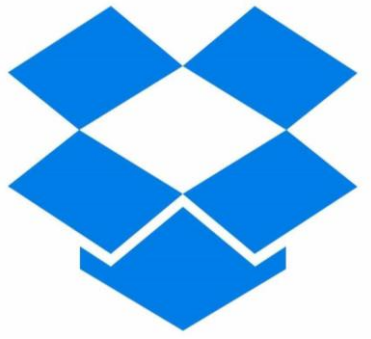


Course Concepts Summary

- Built-in Types, Variables
- Simple & Compound Statements
- Arguments, Functions, Modules
- OOP Concepts, Exceptions
- Advanced Python Topics
- Regular Expressions
- Data Libraries (numpy, pandas, matplotlib)
- Database basics (mysql)
- Web Frameworks (Django)
- Maths (Algebra, Statistics, Probability, Calculus)
- Machine Learning
- Front End (HTML, CSS, JS, BS, React)

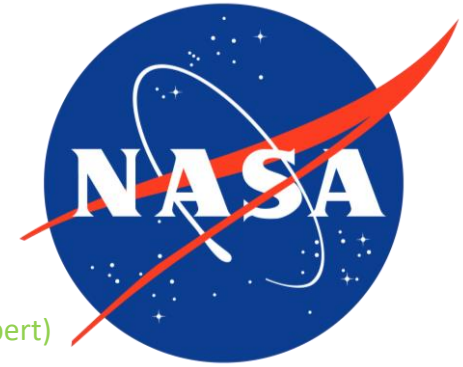
Who using **PYTHON** ?



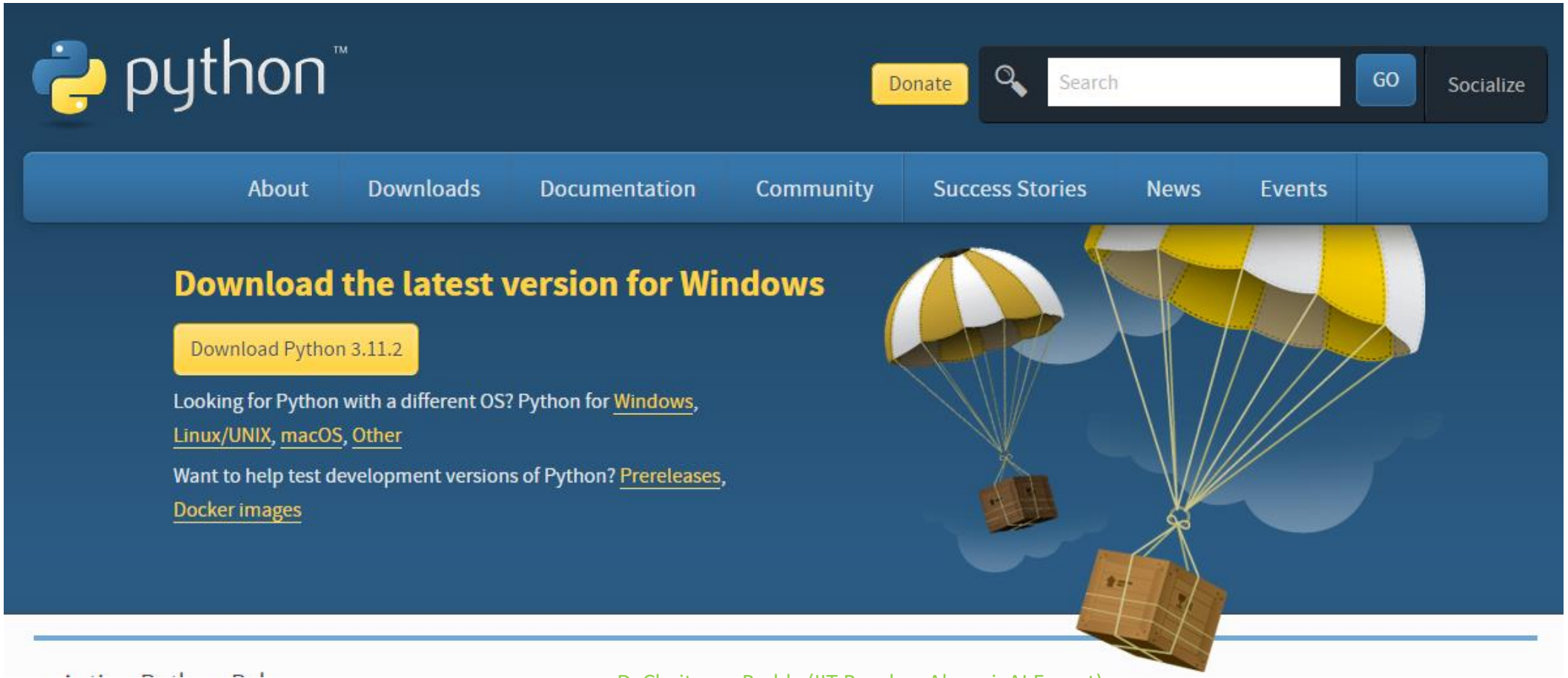




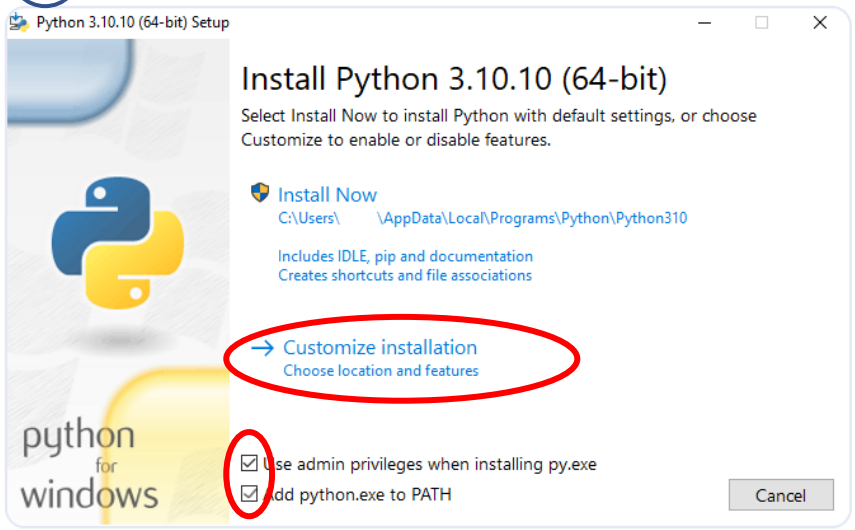

Download **PYTHON**

- *https://www.python.org/downloads/*

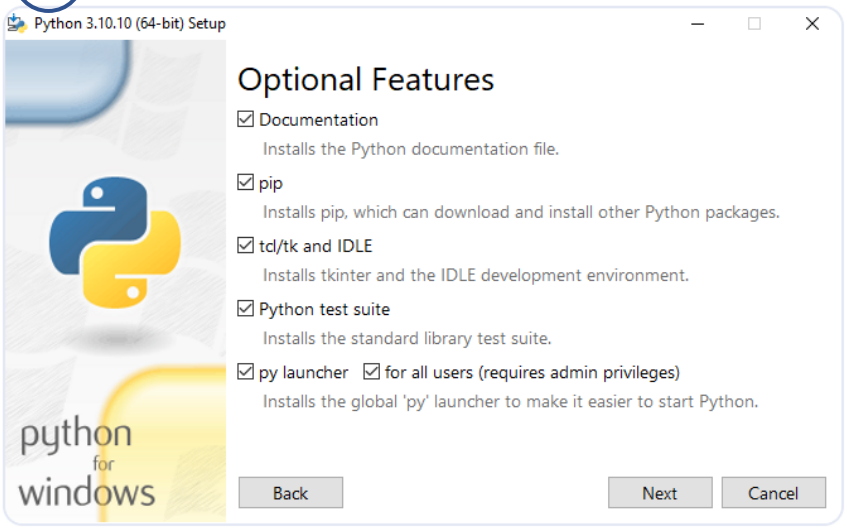


Install **PYTHON**

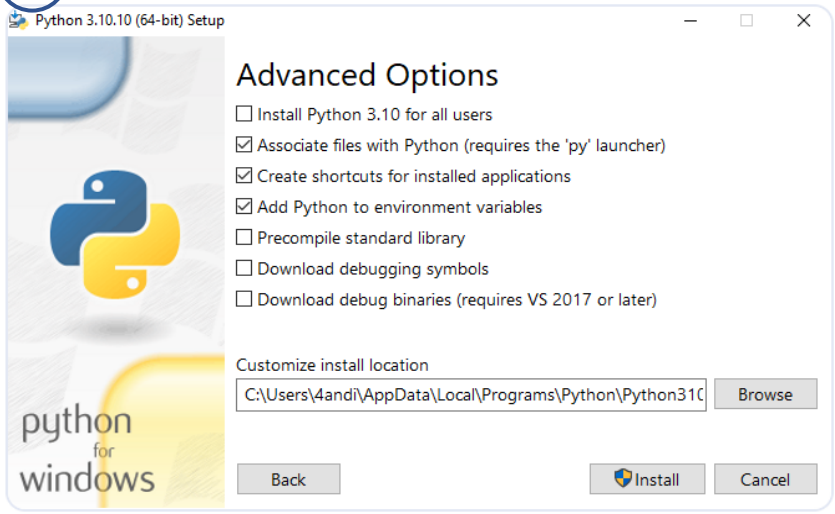
1



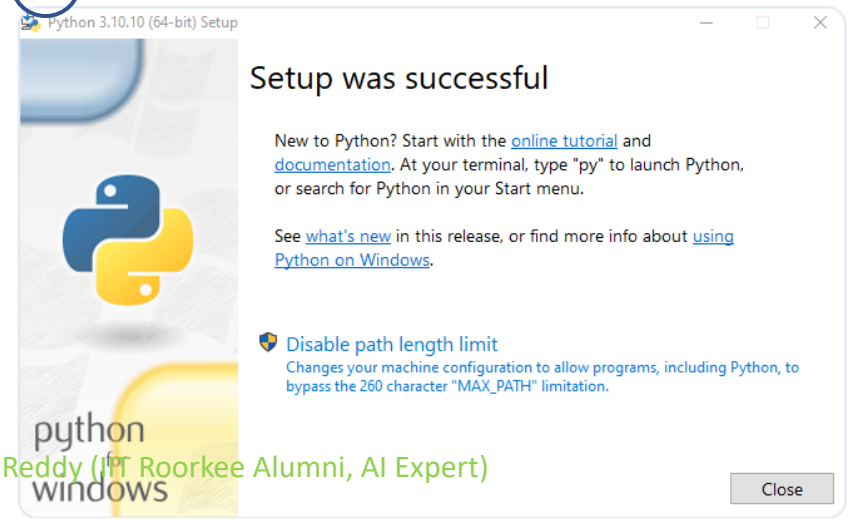
2



3



4

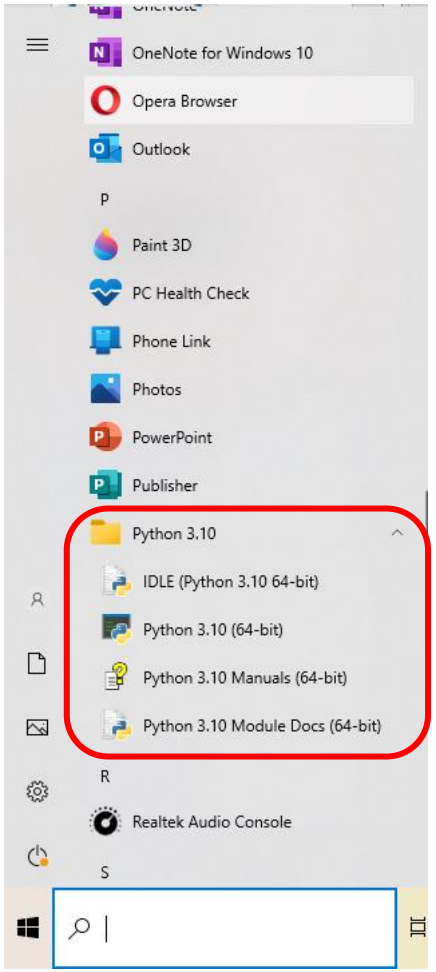


Verify **PYTHON**

In Command prompt:

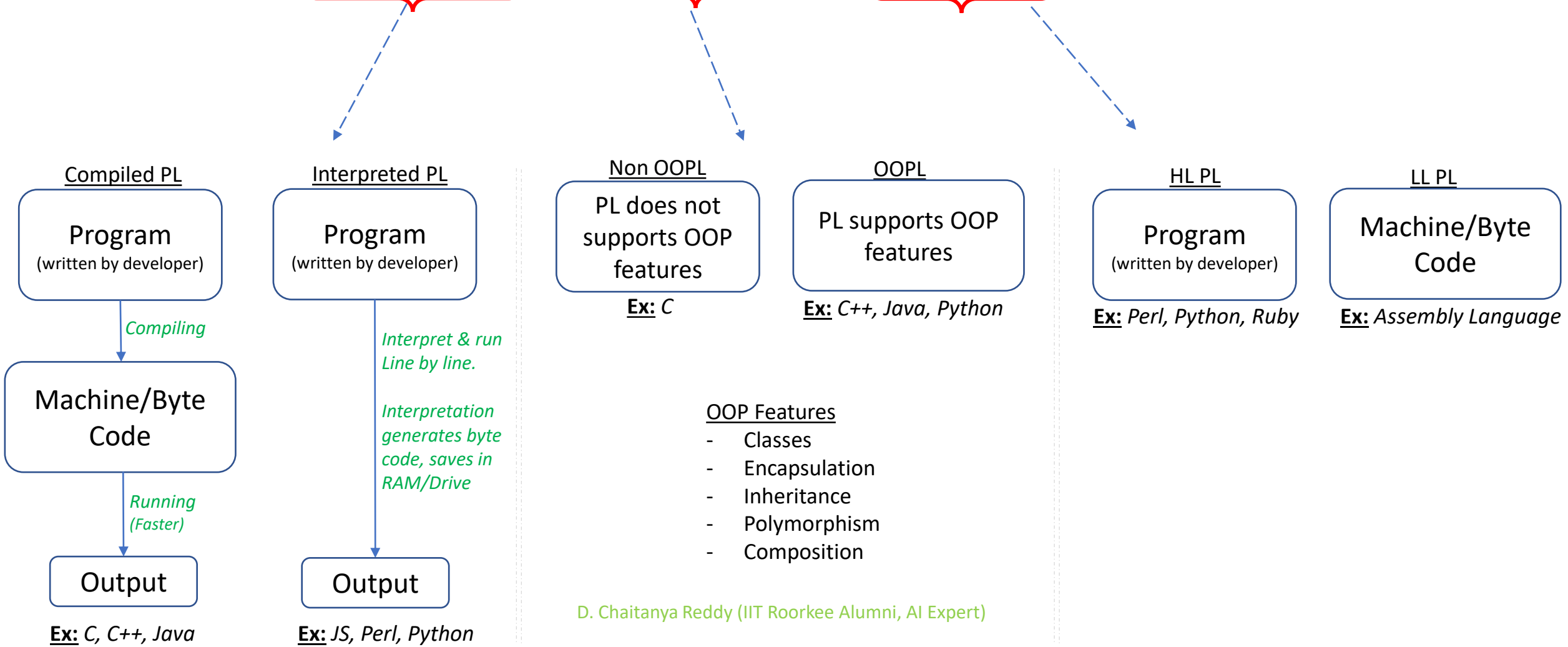
```
C:\Users\LENOVO>python --version
Python 3.10.3

C:\Users\LENOVO>
```

In Start Menu:



What is **PYTHON** ?

- Python is an **interpreted**, **object-oriented**, **high level** programming language.



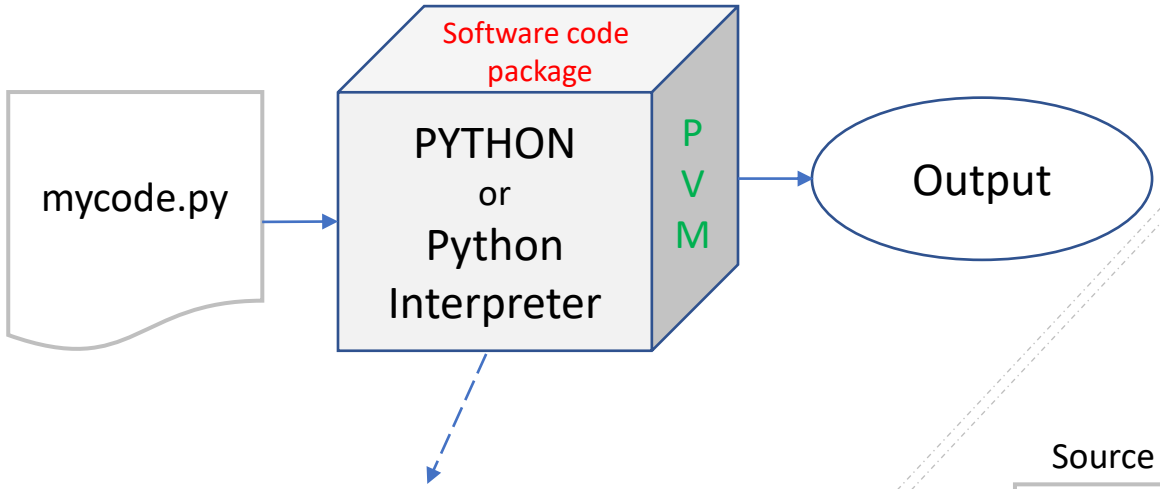
OOP Features
- Classes
- Encapsulation
- Inheritance
- Polymorphism
- Composition

Why **PYTHON** ?

- Developer Productivity
- Portable
- Software Quality
- Easy to Learn
- Huge Support Libraries
- Open Source
- Powerful
- Easy to Use

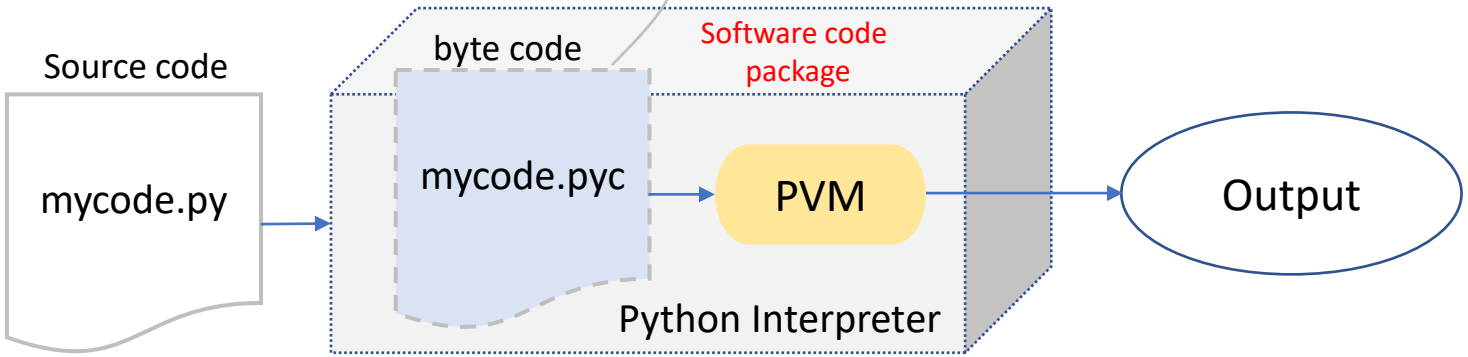
PYTHON is… Interpreter

P V M : **P**ython **V**irtual **M**achine



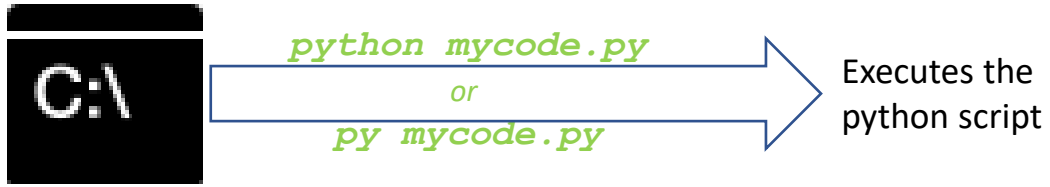
1. Reads line by line of source code (*mycode.py*)
2. Generates byte code for every line (*mycode.pyc*)
3. Executes the byte code by PVM

- **Low level, platform independent** representation of source code.
- Executes much faster

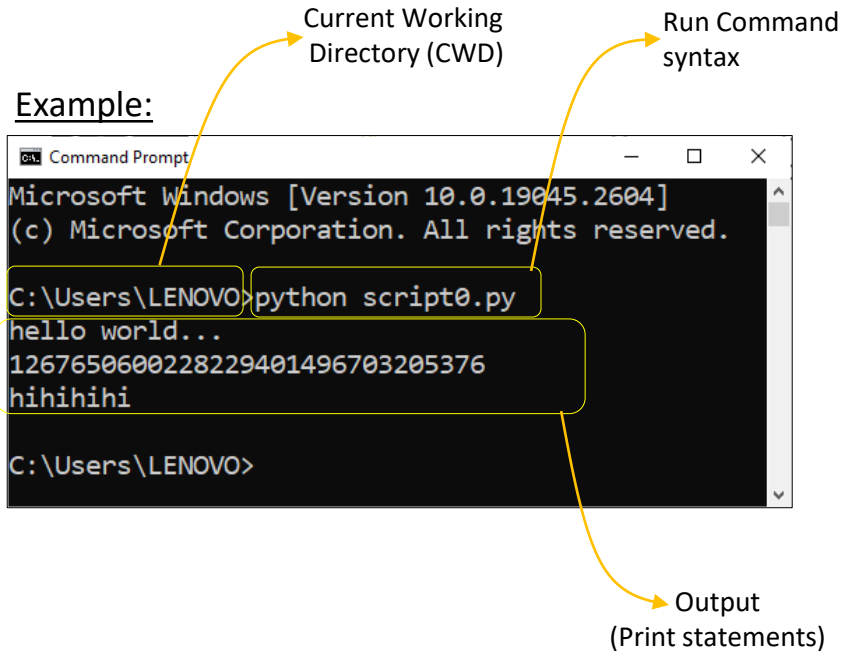


How to run PYTHON script

Using windows command prompt



Example:



`python mycode.py` To get o/p using this command

1) Python installation path must be set in PATH system variable.
2) Source code (mycode.py) must be available in CWD: C:\Users\Lenovo

`python D:\code\mycode.py` To get o/p using this command

1) Python installation path must be set in PATH system variable.
2) NO need to have source code (mycode.py) in CWD.

`py mycode.py` To get o/p using this command

1) NO need to set python installation path in PATH environment variable
2) Source code (mycode.py) must be available in CWD.

`py D:\code\mycode.py` To get o/p using this command

1) NO need to set python installation path in PATH environment variable.
2) NO need to have source code (mycode.py) in CWD.

Built-in Types

Built-in Types, Benefits

Object type	Example literals/creation
Numbers	`1234, 3.1415, 3+4j, 0b111, Decimal(), Fraction()`
Strings	`'spam', "Bob's"`
Lists	`[1, [2, 'three'], 4.5]`
Dictionaries	`{'food': 'spam', 'taste': 'yum'}, dict(hours=10)`
Tuples	`(1, 'spam', 4, 'U'), tuple('spam')`
Files	`open('eggs.txt'), open(r'C:\ham.bin', 'wb')`
Sets	`set('abc'), {'a', 'b', 'c'}`
Other core types	Booleans, types, None
Program unit types	Functions, modules, classes

Why Built-in Types?

- Easy and quick to write programs
- Efficient than custom data structures
- Extensible

Built-in Number objects

Object type	Example literals/creation
Numbers	1234, 3.1415, 3+4j, 0b111, Decimal(), Fraction()
Strings	'spam', "Bob's"
Lists	[1, [2, 'three'], 4.5]
Dictionaries	{'food': 'spam', 'taste': 'yum'}, dict(hours=10)
Tuples	(1, 'spam', 4, 'U'), tuple('spam')
Files	open('eggs.txt'), open(r'C:\ham.bin', 'wb')
Sets	set('abc'), {'a', 'b', 'c'}
Other core types	Booleans, types, None
Program unit types	Functions, modules, classes

Type	Literals	Creating variables: Examples
Integer (unlimited size)	1234, -24, 0, 99999999999999	size = 12, count = 203
Float point number	1.23, 1., 3.14e-10, 4E210, 4.0e+210	percentage = 98.34
Octal	0o177, 0O146	v1 = 0O146
Hex	0x9ff, 0X7AE2	v2 = 0x9ff
Binary	0b1010100, 0B1101	v3 = 0b1010100
Complex number	3+4j, 0+4.0j, 3J	v4 = 3j
Decimal (Fixed Precision)	Decimal('1.0')	d1 = Decimal('1.0')
Fraction (Rational Numbers)	Fraction(1, 3)	f1 = Fraction(1, 3)
Boolean	True, False	exists = False

- Assignment Operator.
- Variables are created and assigned to value when they are first assigned a value.
- Variable values are changed on further assignments.
- Variables are replaced with their values when used in expression.
- Variables must be assigned before they use in expressions.

Integer & Float-point Numbers

Integers

- Decimal digits without decimal point.
- `int(N)` – built in function to convert any number to Integer

Ex:

```
size = 782
print(int(23.43))
```
prints `23`

Float-point Numbers

- Decimal digits with decimal point.
- `float(N)` – built in function to convert any number to float point number

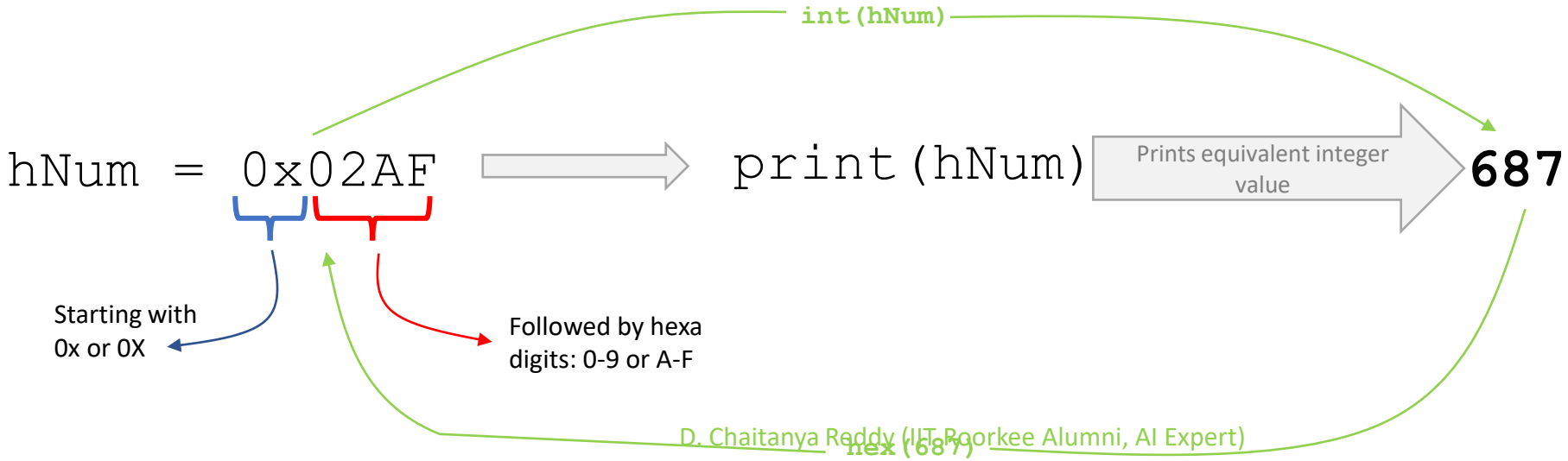
Ex:

```
percentage = 84.23
print(float(size))
```
prints `782.0`

Hexadecimal Numbers

- Integer coded in hexadecimal (base 16)
- Starts with a leading 0x or 0X
- Followed by string of hexadecimal digits 0-9 and A-F
- Hex digits can be coded in lower or upper case.
- `hex(I)` – built in function to convert integer to hex number (**str** representation).
- `int(hex)` – built in function to convert hex number to integer.

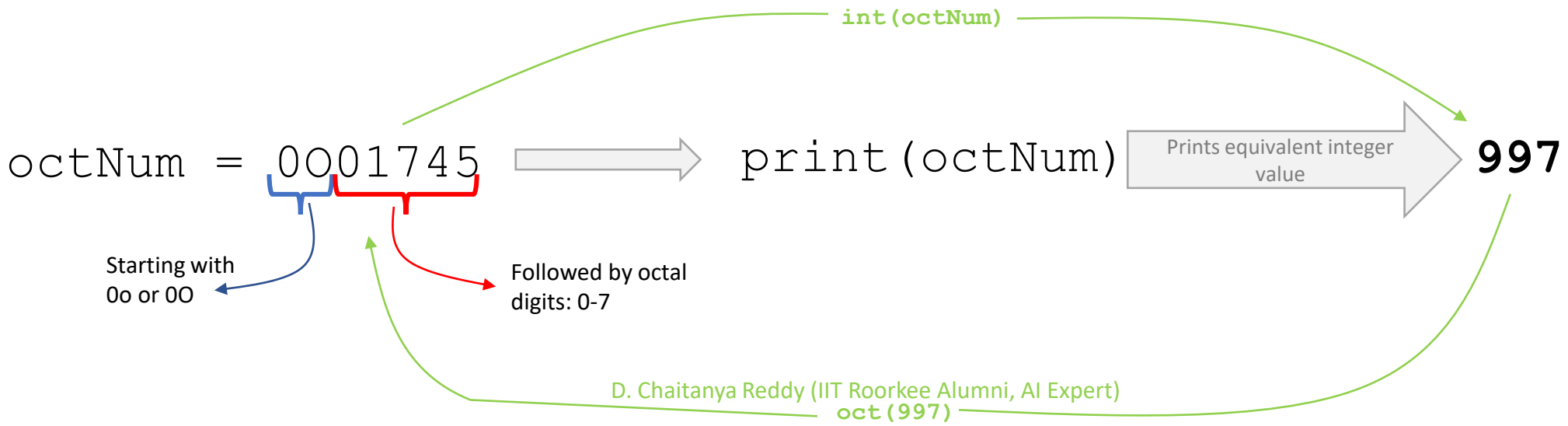
Ex:



Octal Numbers

- Integer coded in octal (base 8)
- Starts with a leading 0o or 0O
- Followed by string of octal digits 0-7
- `oct(I)` – built in function used to convert integer to octal number (**str** repr).
- `int(oct)` – built in function to convert octal number to integer.

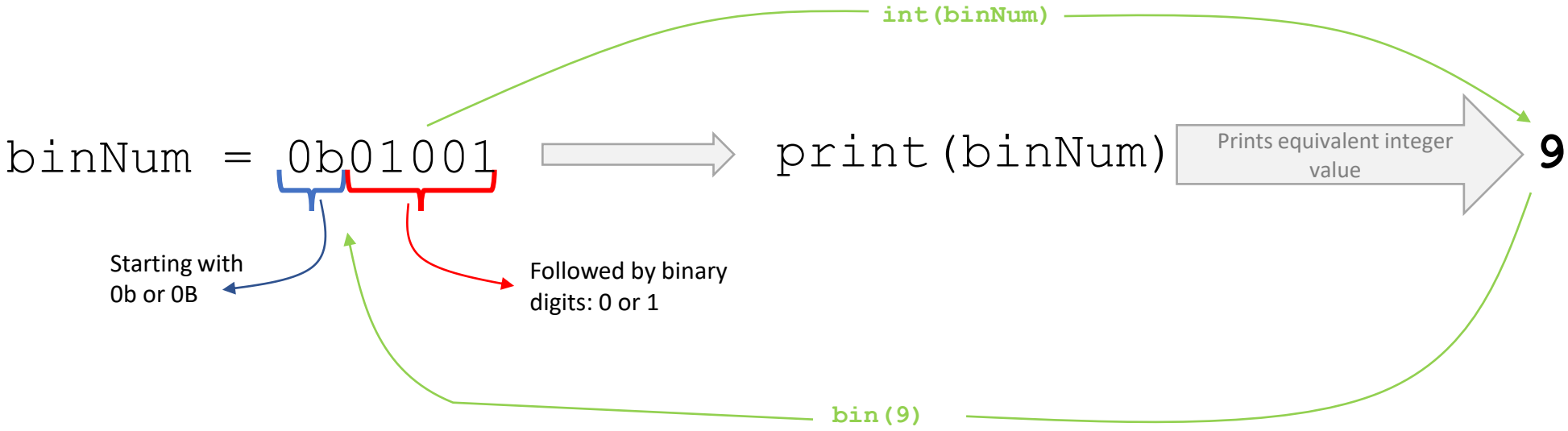
Ex:



Binary Numbers

- Integer coded in binary (base 2)
- Starts with a leading 0b or 0B
- Followed by string of binary digits 0 or 1
- `bin(I)` – built in function used to convert integer to binary number (**str** repr).
- `int(bin)` – built in function to convert binary number to integer.

Ex:



Complex Numbers

- Format: *realpart + imginarypart*
- Real part is optional.
- Imaginary part is terminated with a **`j`** or **`J`**
- Also be created using built in function: `complex(real, img)`

Ex:

```
clxNum = 3.0+4.0j
```

Real part

Imaginary part

created using `complex(real, img)` built in function

```
cNum2 = complex(2, 7)
```

Real part

Imaginary part

Decimals & Fractions

Decimals

- Floating point numbers with fixed number of decimal points.
- Also, called as *fixed-precision* floating point values.
- Well suited for money related calculations

Ex:
```
import decimal
cost = decimal.Decimal('785.42')
decimal.Decimal('1')/decimal.Decimal('7') → Decimal('0.14285714…')
```

Setting decimal precision globally.

```
decimal.getcontext().prec = 3
decimal.Decimal('1')/decimal.Decimal('7') → Decimal('0.143')
```

Can use Integer, float values also

```
decimal.Decimal(1)/decimal.Decimal(7.5) → Decimal('0…')
```

Fractions

- Implements rational numbers
- Keeps both numerator and denominator explicitly to avoid inaccuracies.

Ex:
```
import fractions
x = fractions.Fraction(1, 3)
y = fractions.Fraction(4, 6)
print(x) → 1/3
print(x) → 2/3
x+y prints Fraction(1, 1)
x-y prints Fraction(-1, 3)
print(x-y) → -1/3
x*y prints Fraction(2, 9)
print(x*y) → 2/9
```

Boolean

- Has 2 values: `True`, `False`
- Nothing but integers. but, customized versions of integers: `1` (i.e. `True`), `0` (i.e. `False`)
- Behave exactly as integers `1`, `0`. but, with customized printing logic.

Ex:

```
True + 4 prints 5
True == 1 prints True
k = bool(1)
print(k) prints True
```

Operators for Numbers

Lowest

Operator Precedence Order

Highest

Operator	Meaning
`x < y, x <= y, x > y, x >= y`	Magnitude comparison, Set subset, superset
`x == y, x != y`	Value equality operators
`x | y`	Bitwise OR, Set Union
`x ^ y`	Bitwise XOR
`x & y`	Bitwise AND, Set Intersection
`x << y, x >> y`	Bitwise SHIFT left/right
`x + y`	Addition
`x - y`	Subtraction, Set difference
`x * y`	Multiplication
`x % y`	Remainder
`x / y`	Division: True
`x // y`	Division: Floor
`-x`	Negation
`x ** y`	Power (exponentiation)

1) **`A*B+C*D`** is equal to **`(A*B)+(C*D)`** by applying operator precedence.

2) Can use Paratheses. Ex: `(X+Y)*Z`, `X+(Y*Z)`

3) Mixed types converted up.

Ex: `40+3.14` → (Convert to float) `40.0+3.14` → (Do Addition) `43.14`

4) Normal & Chained Comparisons

Ex: `2.0>=1` → Normal comparison

`1<2>3.0` is equal to `1<2 and 2>3.0` → Chained

(faster) (slower)

5) True Division always keeping remainders in floating point results, regardless of types.

Ex: `10/4` → (results) `2.5`, `-10/4.0` → (results) `-2.5`

6) Floor Division always rounds fractional remainders down to their floor (closest number below true value), regardless of types. Result depends on the types.

Ex: `10//4` → (results) `2`, `-10//4.0` → (results) `-3.0`, `-10//4` → (results) `-3`

Operators for Numbers

Lowest

Operator Precedence Order

Highest

Operator	Meaning
`x < y, x <= y, x > y, x >= y`	Magnitude comparison, Set subset, superset
`x == y, x != y`	Value equality operators
`x | y`	Bitwise OR, Set Union
`x ^ y`	Bitwise XOR
`x & y`	Bitwise AND, Set Intersection
`x << y, x >> y`	Bitwise SHIFT left/right
`x + y`	Addition
`x - y`	Subtraction, Set difference
`x * y`	Multiplication
`x % y`	Remainder
`x / y`	Division: True
`x // y`	Division: Floor
`-x`	Negation
`x ** y`	Power (exponentiation)

7) All bit wise operations shall be performed on equivalent binary numbers and results to integer value.

Ex: `x = 1` *# 1 decimal is 0001 in bits*

`x << 2` prints `4` *# shift left 2 bits: 0100 and prints resulted integer.*

`x | 2` prints `3` *# bitwise OR (either bit=1): 0011*

`x & 1` prints `1` *# bitwise AND (both bits=1): 0001*

8) Other built-in numeric functions: `pow, abs, sum, min, max, …`

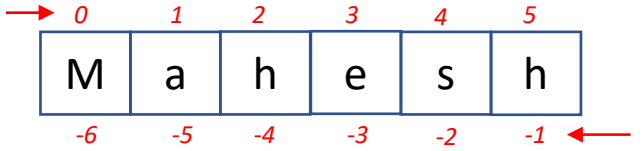
Ex: `pow(2, 4)` → `16`, `abs(-42.0)` → `42.0`

`sum((2, 0, 4))` → `6`, `min(3, 1, 2, 4)` → `1`

9) Other built-in numeric modules: `math, random, round, …`

Ex: `math.floor(-2.567)` → `-3`,
`math.trunc(-2.567)` → `-2`
`math.sqrt(144), math.pi, math.e`
`random.randint(1, 10)` → `5`

Strings



- Used to represent encoded text / bytes.
- **str(name)** to convert any object to string.
- Categorized as immutable sequences.

Object type	Example literals/creation
Numbers	1234, 3.1415, 3+4j, 0b111, Decimal(), Fraction()
Strings	'spam', "Bob's"
Lists	[1, [2, 'three'], 4.5]
Dictionaries	{'food': 'spam', 'taste': 'yum'}, dict(hours=
Tuples	(1, 'spam', 4, 'U'), tuple('spam')
Files	open('eggs.txt'), open(r'C:\ham.bin', 'wb')
Sets	set('abc'), {'a', 'b', 'c'}
Other core types	Booleans, types, None
Program unit types	Functions, modules, classes

Different String versions:

String Version	Interpretation
`S = '' or S = str()`	Empty String
`S = "spam's"` `S = 'spam"s'`	Double quotes, same as single
`S = 's\npa\tm'`	Escape sequences
`S = """…multiline…"""`	Triple quoted block strings
`S = r'c:\temp\new'`	Raw string
`B = b'sp\xc4m'`	Byte string

String basic operations:

Operation	Interpretation
`S1 + S2`	Concatenate
`S * 3`	Repeat
`S[i], S[-i]`	Indexing
`S[i:j], S[i:], S[i:j:k], S[::k]`	Slicing, slicing with skip
`len(S)`	Length of string
`'a %s parrot %s' % (S1, S2)`	String formatting expr
`S in 'spam'`	Substring finding

Sequence operations

1. Multi line text in code.

```
menu = """spam
eggs
drinks
"""
```

2. Documentation purpose.
3. Disabling/commenting the code.

Mutable: In place changes are allowed for object's value
Immutable: In place changes are NOT allowed for object's value
Sequence: object that support sequence operations

Strings

String methods

String method	Interpretation
`'a {0} parrot'.format(S1)`	String formatting method
`S.find('pa')`	String search
`S.rstrip()`	Remove whitespaces
`S.replace('pa', 'xx')`	Replacement
`S.split(',')`	Split with char
`S.isdigit()`	Content test
`S.lower()`	Case conversion
`S.endswith()`	End test
`','.join(strlist)`	Join with char

Any operation/method call does NOT change the string, rather generates a new string object.

Escape chars

Escape	Meaning
`\\`	Single backslash
`\'`	Single quote
`\"`	Double quote
`\n`	New line
`\t`	Horizontal tab
`\v`	Vertical tab

ASCII values: (0 – 127)

Char	ASCII Value
`A - Z`	65 to 90
`a - z`	97 to 122
`0 - 9`	48 to 57
`Space`	32

- Char to ASCII: ord()

  `ord('s')` → `115`

- ASCII to Char: chr()

  `chr(115)` → `'s'`

Iterating String Sequence

1) Easy and Quick to write code.
2) 10x Faster in execution.
3) Saves memory

```
name = 'Mahesh Babu'
```

Using for loop:

```
name = 'Mahesh Babu'
new_name = ''
for ch in name:
    new_name = new_name + ch*2

print(new_name)
```

Output:

```
MMaahheesshh  BBaabbuu
```

Using List Comprehension:

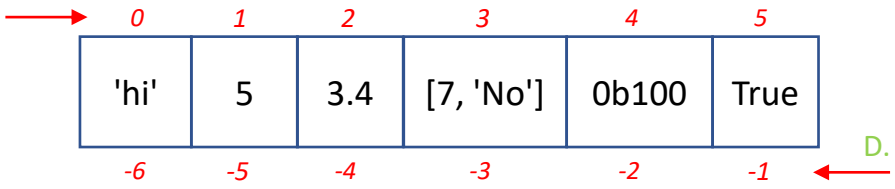
```
''.join([ch*2 for ch in name])
```

Output:

```
MMaahheesshh  BBaabbuu
```

Lists

- An **ordered** (insertion order) collection of **arbitrary** objects.
- Accessed by offset/index, i.e., supports sequence operations.
- **Mutable**: can grow and shrink on demand.
- **Iterable**: can traverse through all elements
- **Nestable**: can create list of list of list, so on.
- **Heterogeneous**: can contain any type of object as elements.



Object type	Example literals/creation
Numbers ✓	1234, 3.1415, 3+4j, 0b111, Decimal(), Fraction()
Strings ✓	'spam', "Bob's"
Lists	[1, [2, 'three'], 4.5]
Dictionaries	{'food': 'spam', 'taste': 'yum'}, dict(hours=10)
Tuples	(1, 'spam', 4, 'U'), tuple('spam')
Files	open('eggs.txt'), open(r'C:\ham.bin', 'wb')
Sets	set('abc'), {'a', 'b', 'c'}
Other core types	Booleans, types, None
Program unit types	Functions, modules, classes

List creation techniques:

List Version	Interpretation
`L = [] or L = list()`	Empty list
`L = [3, 'a', 5.4, {}, [4, 5]]`	List with 5 items with nesting
`L = list('spam')`	List of an iterable's items
`L = list(range(-4, 4))`	List of successive integers

List basic operations:

Sequence operations

Operation	Interpretation
`L[i], L[-i], L[i][j], L[i:j], L[-i:]`	Index, index of index, slicing
`L1 + L2`	Concatenate
`L * 3`	Repeat
`len(L)`	Length/size of list
`3 in L`	Element existence test/finding

Results new list, NO in place changes

Lists

List methods

List method	Interpretation
`L.append(4)`	Add new element at last.
`L.extend([5, 1.23, 'hi'])`	Add multiple elements at last.
`L.insert(i, el)`	Add element at index i.
`L.index(el)`	Get the first index (from left) of element el.
`L.count(el)`	Get the count of element el.
`L.sort()`	Sort elements.
`L.reverse()`	Reverse the order of elements.
`L.copy()`	Copy the list.
`L.clear()`	Remove all elements in list.
`L.pop(i)`	Remove and return element at index i.
`L.remove(el)`	Remove the element el at all indexes.
`del L[i], del L[i:j]`	Delete element at index i, elements in slice `i:j`
`L[i:j]=[]`	Removing all elements in slice `i:j`
`L[i] = 3, L[i:j]=[4, 5, 6]`	Index assignment, slice assignment.

Any operation/method call results in place change of the list, rather generating a new list.

Iterating List Sequence

1) Easy and Quick to write code.
2) 10x Faster in execution.
3) Saves memory

```
my_list = [1, 'hi', 3.4, [6, 7]]
```

Using for loop:

```
my_list = [1, 'hi', 3.4, [6, 7]]
new_list = []
for el in my_list:
    new_list.append(el*2)

print(new_list)
```

Output:

```
[2, 'hihi', 6.8, [6, 7, 6, 7]]
```

Using List Comprehension:

```
[el*2 for el in my_list]
```

Output:

```
[2, 'hihi', 6.8, [6, 7, 6, 7]]
```

Tuples

- An **ordered** (insertion order) collection of **arbitrary** objects.
- Accessed by offset/index, i.e., supports sequence operations.
- **Immutable**: fixed length. Applicable only to top level. Not to it's content.
- **Iterable**: can traverse through all elements
- **Nestable**: can create tuple of tuple of tuple, so on.
- **Heterogeneous**: can contain any type of object as elements.

Object type	Example literals/creation
Numbers ✓	`1234, 3.1415, 3+4j, 0b111, Decimal(), Fraction()`
Strings ✓	`'spam', "Bob's"`
Lists ✓	`[1, [2, 'three'], 4.5]`
Dictionaries	`{'food': 'spam', 'taste': 'yum'}, dict(hours=10)`
Tuples	`(1, 'spam', 4, 'U'), tuple('spam')`
Files	`open('eggs.txt'), open(r'C:\ham.bin', 'wb')`
Sets	`set('abc'), {'a', 'b', 'c'}`
Other core types	Booleans, types, None
Program unit types	Functions, modules, classes

Lists vs Tuples

Tuple creation techniques:

Tuple Version	Interpretation
`T = (), T = tuple()`	Empty tuple
`T = (4, )` ~~`T = (4)`~~	Tuple with single element
`T = (0, 3.4, 'hi', ('bye', 1))`	Tuple with 4 elements and nested tuple
`T = tuple('spam')`	Tuple of items in an Iterable.

Tuple basic operations:

Sequence operations

Operation	Interpretation
`T[i], T[-i], T[i][j], T[i:j], T[-i:]`	Index, index of index, slicing
`T1 + T2`	Concatenate
`T * 3`	Repeat
`len(T)`	Length/size of tuple
`'spam' in T`	Element existence test/finding

Results new tuple, NO in place changes

Tuple methods

List method	Interpretation
`T.index(el)`	Get the first index (from left) of element el.
`T.count(el)`	Get the count of element el.

Iterating Tuple Sequence

1) Easy and Quick to write code.
2) 10x Faster in execution.
3) Saves memory

```
my_tup = (1, 'hi', 3.4, [6, 7])
```

Using for loop:

```
my_tup = (1, 'hi', 3.4, [6, 7])
new_list = []
New_tup = ()
for el in my_tup:
    new_list.append(el*2)

new_tup = tuple(new_list)
print(new_tup)
```

Output:

```
(2, 'hihi', 6.8, [6, 7, 6, 7])
```

Using List Comprehension:

```
tuple([el*2 for el in my_tup])
```

Output:

```
(2, 'hihi', 6.8, [6, 7, 6, 7])
```

Sets

- An **unordered** collection of **unique** and **immutable** objects.
- Designed to support operations corresponding to mathematical **set** theory.
- **Mutable**: can grow and shrink on demand.
- **Iterable**: can traverse through all elements
- All standard operations of Set uses efficient algorithms and hence faster results

Object type	Example literals/creation
Numbers ✓	1234, 3.1415, 3+4j, 0b111, Decimal(), Fraction()
Strings ✓	'spam', "Bob's"
Lists ✓	[1, [2, 'three'], 4.5]
Dictionaries	{'food': 'spam', 'taste': 'yum'}, dict(hours=10)
Tuples ✓	(1, 'spam', 4, 'U'), tuple('spam')
Files	open('eggs.txt'), open(r'C:\ham.bin', 'wb')
Sets	set('abc'), {'a', 'b', 'c'}
Other core types	Booleans, types, None
Program unit types	Functions, modules, classes

Ex:

```
s=set()
s1=set([1, 7.4, True, 5])   #built-in call for set creation
s2={1, 'sp', 7, 'e'}        #new literal for set creation
s3=set('spa')
s3  prints  {'s', 'a', 'p'}
s3.add('h'),  s3  prints  {'s', 'h', 'a', 'p'}
s1&s2  prints  {1}
s1|s2  prints  {1, 5, 7, 7.4, 'e', 'sp'}
s1-s2  prints  {5, 7.4}
s1^s2  prints  {5, 7, 7.4, 'e', 'sp'}
s1.union(s2) is same as s1|s2
s1.intersection(s2) is same as s1&s2
```

Mutable: value of the object can't be changed in place.
Immutable: In place changes are allowed for object's value.

Sets

- **Frozenset**: Same as **Set** behavior except immutable in nature.

  Ex: `s4 = frozenset([1, 5.6, 'rt', True])`

- Can be used to *filter duplicates* out of other collections.

```
rl1 = random.sample(range(10, 20), 10)
```

- Can be used to *isolate differences* in other iterables like lists, strings, etc…

- Can be used to perform *order neutral equality* test.

- More *suitable* for *large datasets* if order does not matter to get faster results

Iterating Set

```
myset = {1, 7, 8, 9, 0, 5, -1}
```

1) Easy and Quick to write code.
2) 10x Faster in execution.

Using for loop:

```
newSet = set()
for num in myset:
    newSet.add(num*2)
print(newSet)
```

Output:

```
{0, 2, 10, 14, 16, 18, -2}
```

Using Set comprehension:

```
{num*2 for num in myset}
```

Output:

```
{-2, 0, 2, 10, 14, 16, 18}
```

Dictionaries

Object type	Example literals/creation
Numbers ✓	1234, 3.1415, 3+4j, 0b111, Decimal(), Fraction()
Strings ✓	'spam', "Bob's"
Lists ✓	[1, [2, 'three'], 4.5]
Dictionaries	{'food': 'spam', 'taste': 'yum'}, dict(hours=10)
Tuples ✓	(1, 'spam', 4, 'U'), tuple('spam')
Files	open('eggs.txt'), open(r'C:\ham.bin', 'wb')
Sets ✓	set('abc'), {'a', 'b', 'c'}
Other core types	Booleans, types, None
Program unit types	Functions, modules, classes

- Every element/item is in **key**, **value** pair format.
- An **ordered** (from 3.7 version) collection of **arbitrary** objects.
- Accessed by key, instead of offset/index.
- **Indexing** (fetching element) is very fast.
- **Mutable**: can grow and shrink on demand.
- **Iterable**: can traverse through all elements
- **Nestable**: can contain dict in another dict.
- **Heterogeneous**: can contain any type of objects.
- Does not support sequence operations.

`dict` maintains **unique keys**

Key: not only a string, any immutable objects allowed as key.
Value: any arbitrary object is allowed as value.

Dictionary creation techniques:

Dictionary Version	Interpretation
`D = {}, D = dict()`	Empty dictionary
`D = {'name': 'Mahesh', 'age':40}`	Simple creation
`E = {'CTO': {'name': 'Mahesh', 'age':40}}`	Nested dictionary
`D = dict(name='Mahesh', age=40)`	Alternate creation
`D = dict([('name', 'Mahesh'), ('age', 40)])`	Creation with key value pairs in list
`D = dict(zip(keylist, valuelist))`	Creation with zipped key value lists
`D = dict.fromkeys(['name', 'age'], default?)`	Creation using keys and values with default (or **None**).

The keys are by default string. No other types are allowed for keys in this technique.

Dictionaries

Dictionary methods

Dictionary method	Interpretation
`D['name']`	Indexing (fetching element) by key
`E['CTO']['age']`	Indexing (fetching element) by keys in nested dictionaries.
`'age' in D`	key existence test/finding
`D.keys()`	Dictionary keys view
`D.values()`	Dictionary values view
`D.items()`	Dictionary items (key, value pairs) view
`D.copy()`	Copy dictionary
`D.clear()`	Remove all items in dictionary
`D.update(D2)`	Merge by keys. Overwrites values of same key if clash found. `D2` overwrites `D` here.
`D.get(key, default?)`	Fetch by key. If absent, default (or None)
`D.pop(key, default?)`	Remove by key. If absent, default (or error)
`len(D), sorted(D)`	Length of dictionary (number items), sorting dictionary items by keys
`D[key]=42`	Adding/Changing value by key
`del D[key]`	Deleting item by key

Key: not only a string, any immutable object allowed for key.

Value: any arbitrary object is allowed for value.

- Views also retain original order of dictionary.
- Reflect future changes to the dictionary
- Support set operations.

Any operation/method call results in place change of the dictionary, rather generating a new dictionary.

Creating Dictionary using Comprehension

```
key_list = ['name', 'profression', 'age', 'location']
value_list = ['Allu Arjun', 'hero', 35, 'hyd']
```

1) Easy and Quick to write code.

2) 10x Faster in execution.

3) Saves memory

Using for loop:

```
my_dict = {}
for indx in range(len(key_list)):
        my_dict[key_list[indx]]=value_list[indx]

print(my_dict)
```

Output:

```
{'name': 'Allu Arjun', 'profression': 'hero',
'age': 35, 'location': 'hyd'}
```

Using dictionary Comprehension:

```
{key: value for (key, value) in zip(key_list,
value_list)}
```

Output:

```
{'name': 'Allu Arjun', 'profression': 'hero',
'age': 35, 'location': 'hyd'}
```

Iterating Dictionary mapping

```
my_dict = {'name': 'Allu Arjun', 'profression': 'hero', 'age': 35, 'location': 'hyd'}
```

Using for loop:

```
for key in my_dict.keys():
        if key!='age':
                print(key, my_dict[key])
```

```
for key in my_dict:
        if key!='age':
                print(key, my_dict[key])
```

```
for (key,value) in my_dict.items():
        if key!='age':
                print(key, value)
```

Output:

```
name Allu Arjun
profression hero
location hyd
```

Using dictionary Comprehension:

```
{key:my_dict[key] for key in my_dict.keys() if key!='age'}
```


```
{key:my_dict[key] for key in my_dict.keys() if key!='age'}
```


```
{key:value for (key,value) in my_dict.items() if key!='age'}
```

Output:

```
{'name': 'Allu Arjun', 'profression': 'hero', 'location': 'hyd'}
```

1) Easy and Quick to write code.
2) 10x Faster in execution.

Named Tuples

- Same as tuple, except accessible by offset/index and/or keys.

```
>>> bob = ('Bob', 40.5, ['dev', 'mgr'])              #tuple record
>>> bob[0], bob[2]                                  # accessible by position
('Bob', ['dev', 'mgr'])
```

```
>>> bob = dict(name='Bob', age=40.5, jobs=[['dev', 'mgr'])          # dictionary record
>>> bob[name], bob['jobs']                                          # accessible by key
('Bob', ['dev', 'mgr'])
```

```
>>> from collections import namedtuple
>>> Rec = namedtuple('my_nt', ['name', 'age', 'jobs'])
>>> bob = Rec('Bob', 40.5, ['dev', 'mgr'])                    # dictionary record
>>> bob[0], bob[2]                                            # accessible by position
('Bob', ['dev', 'mgr'])
>>> bob.name, bob.age                                         # accessible also by key/name
('Bob', 40.5)
```

Files

- To deal with files (reading, writing files)
- The built-in **open** function creates a python file object.
- Serves as a link to a file residing on the machine.

File creation techniques:

File creation	Interpretation
`op = open(r'C:\temp\spam.txt', 'w')`	Create output file (`'w'` means write)
`ip = open('ham.txt', 'r')` `Ip = open(r'C:\temp\ham.txt')`	Create input file (`'r'` means read). Default mode is `'r'`

File methods

File method	Interpretation
`aStr = ip.read()`	Read entire file into a string
`aStr = ip.read(N)`	Read upto next N chars (or bytes) into a string
`aStr = ip.readline()`	Read next line (including `\n` newline) into a string
`aList = ip.readlines()`	Read entire file into list of strings (with `\n`)
`op.write(aStr)`	Write a string of chars (or bytes) into file.

Object type	Example literals/creation
Numbers ✓	`1234, 3.1415, 3+4j, 0b111, Decimal(), Fraction()`
Strings ✓	`'spam', "Bob's"`
Lists ✓	`[1, [2, 'three'], 4.5]`
Dictionaries ✓	`{'food': 'spam', 'taste': 'yum'}, dict(hours=10)`
Tuples ✓	`(1, 'spam', 4, 'U'), tuple('spam')`
Files	`open('eggs.txt'), open(r'C:\ham.bin', 'wb')`
Sets ✓	`set('abc'), {'a', 'b', 'c'}`
Other core types	Booleans, types, None
Program unit types	Functions, modules, classes

File methods (contd…)

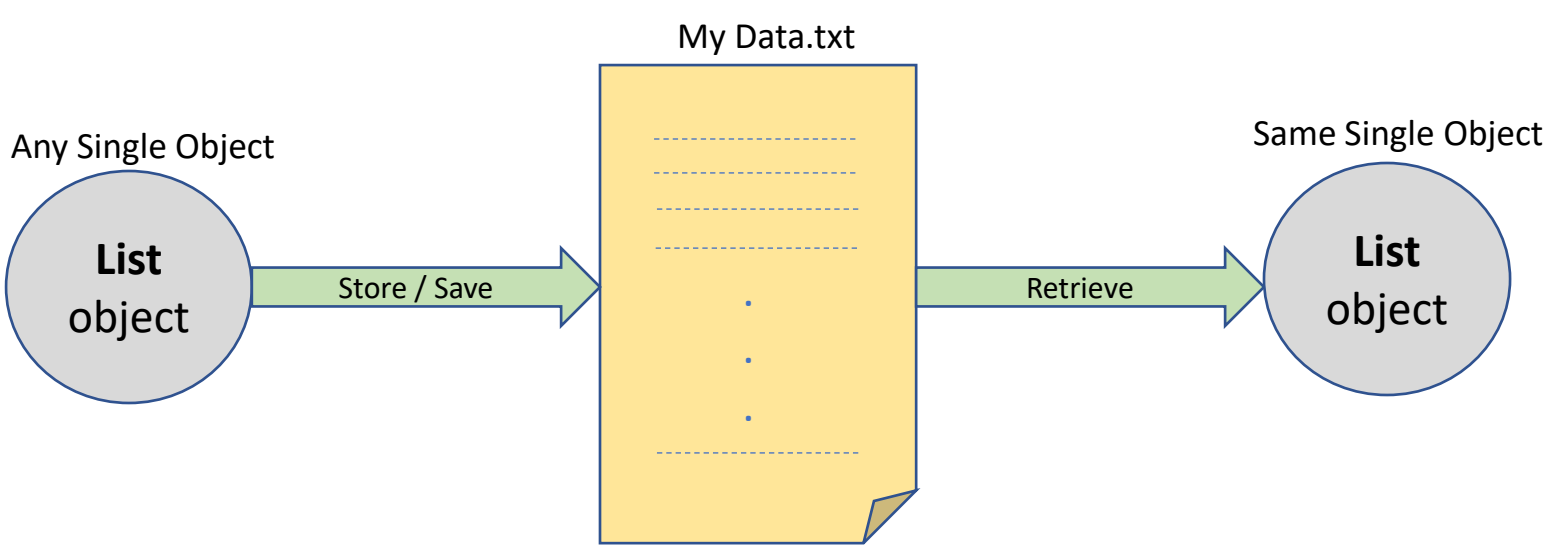
File method	Interpretation
`op.writelines(aList)`	Write all line strings in a list into file.
`op.close()`	Manual closing file.
`op.flush()`	Flush output buffer into disk.
`any_file.seek(N)`	Change file position to offset N for next operation.

File processing modes:

mode	Interpretation
`r`	Read a text file.
`w`	Write to a text file.
`a`	Append text to end of file.
`rb`	Read binary file.

mode	Interpretation
`wb`	Write to binary file.
`+`	Read & write to same file.

Storing objects in Files



Pickle (using **pickle** module)

Shelve (using **shelve** module)

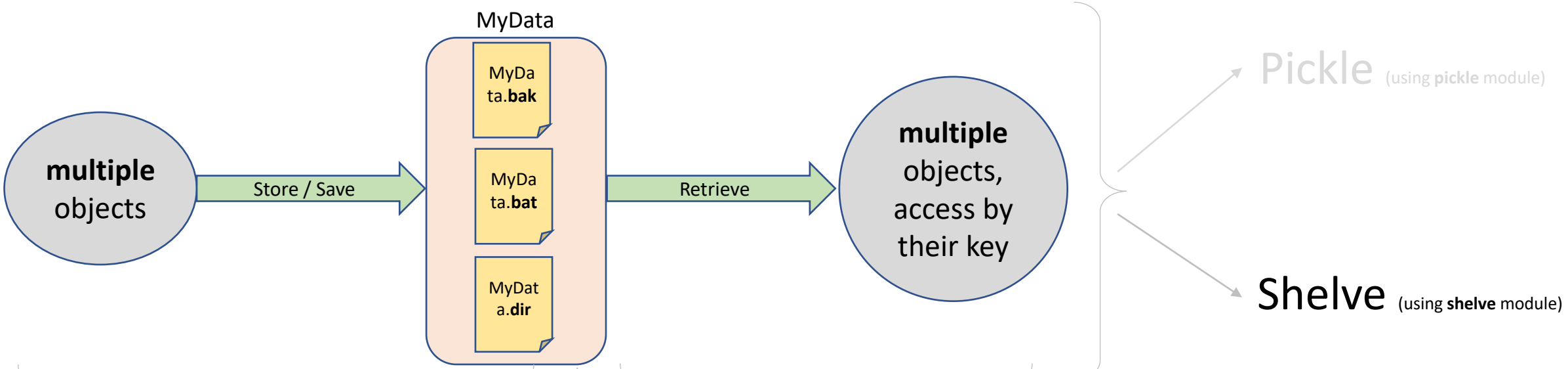
```
my_list=[1, 'hi', 3.4, [6, 7]]
import pickle
F = open(r'C:\tmp\My Data.txt', 'wb')
pickle.dump(my_list, F)
F.close()
```

```
my_file = open(r'C:\tmp\My Data.txt', 'rb')
saved_obj = pickle.load(my_file)
print(saved_obj)
```

```
Output:
[1, 'hi', 3.4, [6, 7]]
```

Pickling & Unpickling

Storing objects in Files




```
my_list=[1, 'hi', 3.4, [6, 7]]

my_map={'name': 'Allu Arjun', 'profression':
'hero', 'age': 35, 'location': 'hyd'}

my_tup=('Bob', 40.5, ['dev', 'mgr'])

import shelve

my_data = shelve.open('MyData')
```

Creates these 3 files in CWD

```
my_data['list_obj']=my_list

my_data['allu_info']=my_map

my_data['bob_details']=my_tup

my_data.close()
```

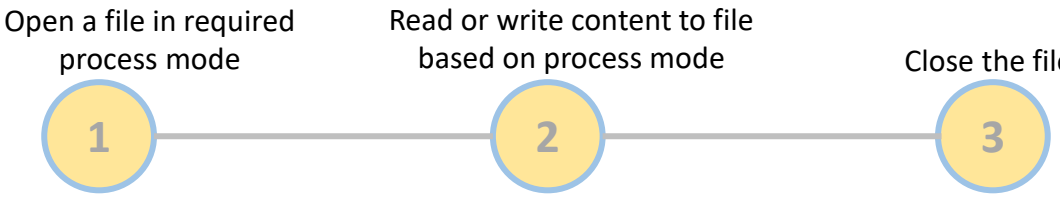
Writes object information to 3 files

```
my_shelve = shelve.open('MyData')

my_saved_list = my_shelve['list_obj']

my_saved_map = my_shelve['allu_info']

my_saved_tup = my_shelve['bob_details']

print(my_saved_list, my_saved_map, my_saved_tup)
```

```
Output:

[1, 'hi', 3.4, [6, 7]] {'name': 'Allu Arjun', 'profression':
'hero', 'age': 35, 'location': 'hyd'} ('Bob', 40.5, ['dev',
'mgr'])
```

Shelve Technique

File Context Managers



Error prone code

```
1 my_file = open(r'C:\temp\hello.txt', 'r')
2 for line in my_file:
        print(line[:5])
3 my_file.close()
```

Corrected code

```
1 my_file = open(r'C:\temp\hello.txt', 'r')
2 try:
        for line in my_file:
            print(line[:5])
3 finally:
        my_file.close()
```

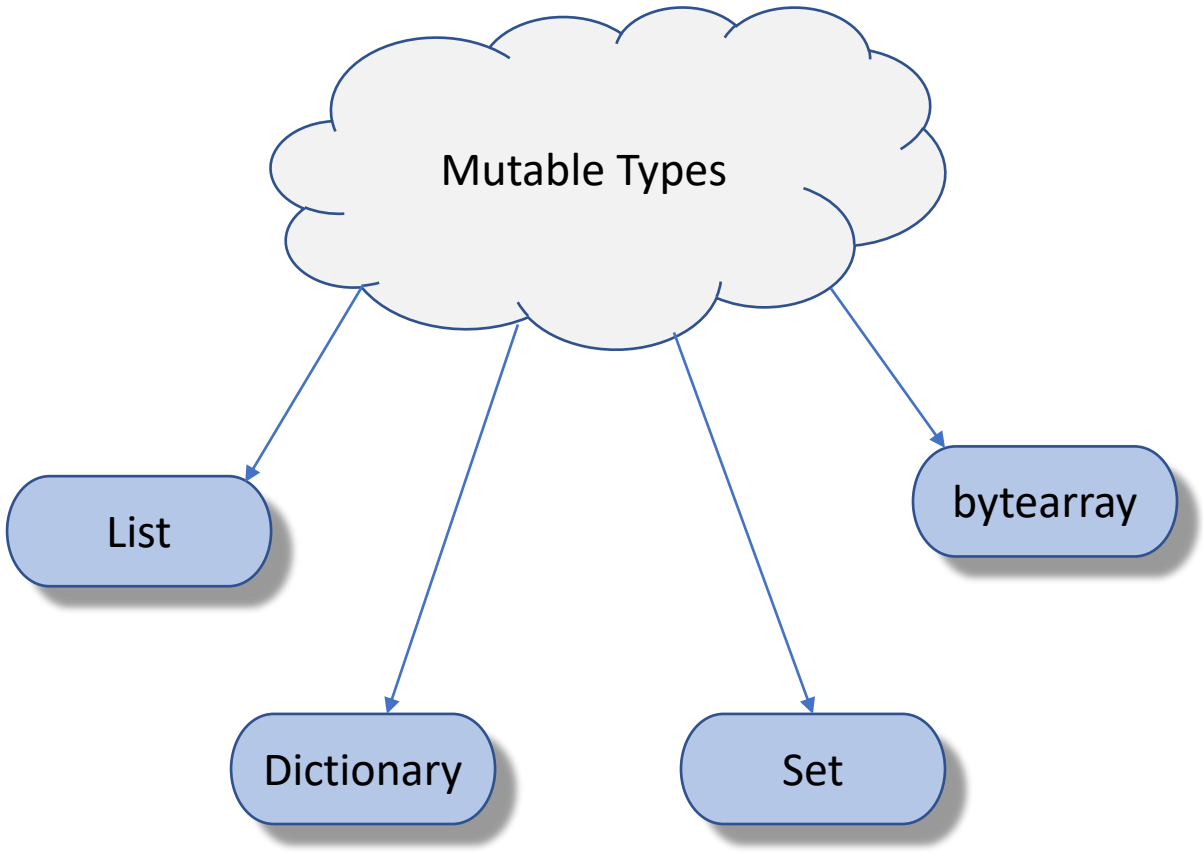
Simplified Corrected code

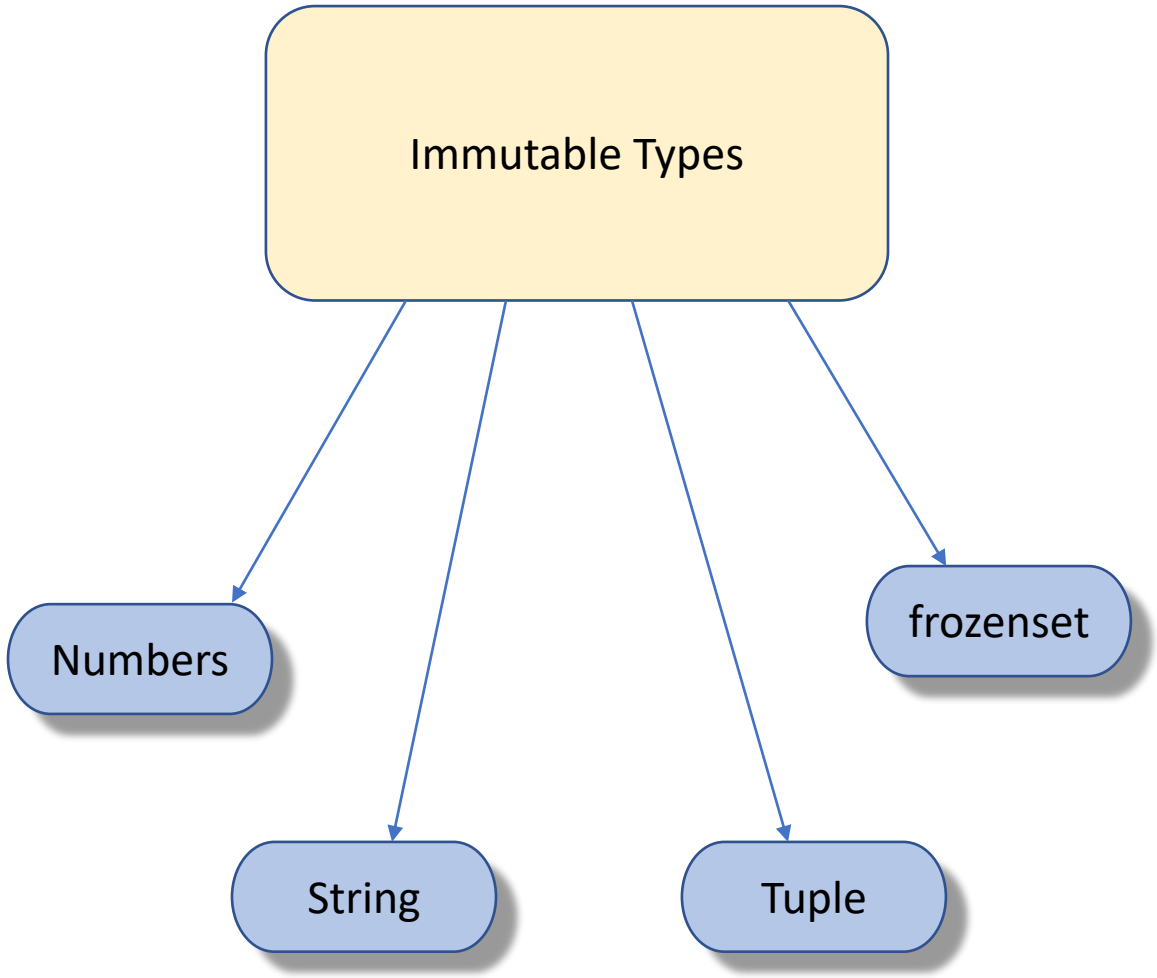
```
1 3 with open(r'C:\temp\hello.txt', 'r') as my_file:
2       for line in my_file:
            print(line[:5])
```

Built-in Types Summary

Object Type	Category	Ordered?	Mutable?	Nestable?	Elements Unique?
All Numbers	Numeric	N/A	No	N/A	N/A
Strings	Sequence	Yes	No	N/A	No
Lists	Sequence	Yes	Yes	Yes	No
Dictionaries	Mapping	Yes (from 3.7)	Yes	Yes	No
Tuples	Sequence	Yes	No	Yes	No
Sets	Set	No	Yes	No	Yes
Frozen sets	Set	No	No	Yes	Yes
Files	Extension	N/A	N/A	N/A	N/A
`bytearray`	Sequence	Yes	Yes	N/A	No

Built-in Types Summary





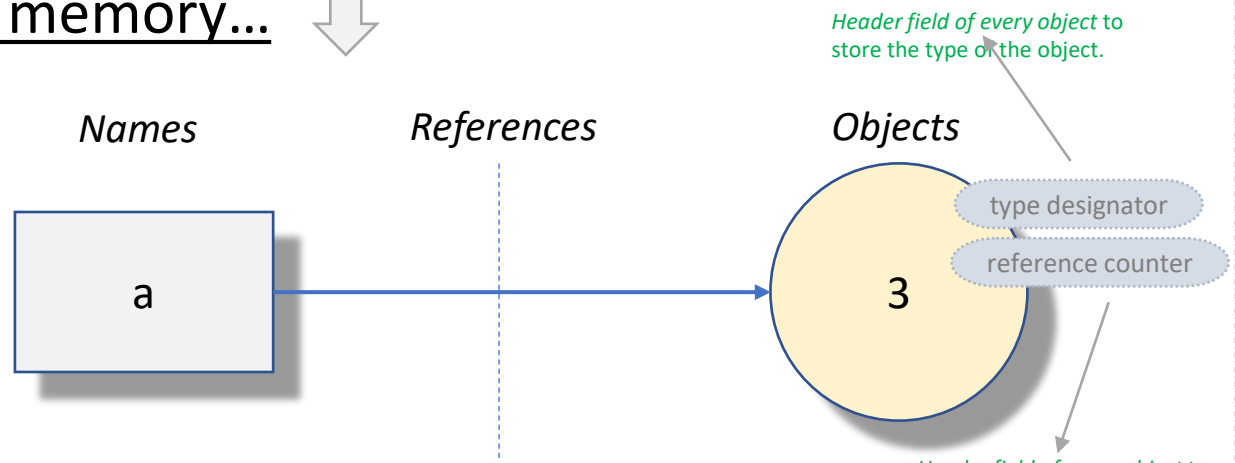
Garbage Collection

Recap….

```
a = 3
```

- Assignment Operator.
- Variables are created and assigned to value when they are first assigned a value.
- Variable values are changed on further assignments.
- Variables are replaced with their values when used in expression.
- Variables must be assigned before they use in expressions.

In memory…

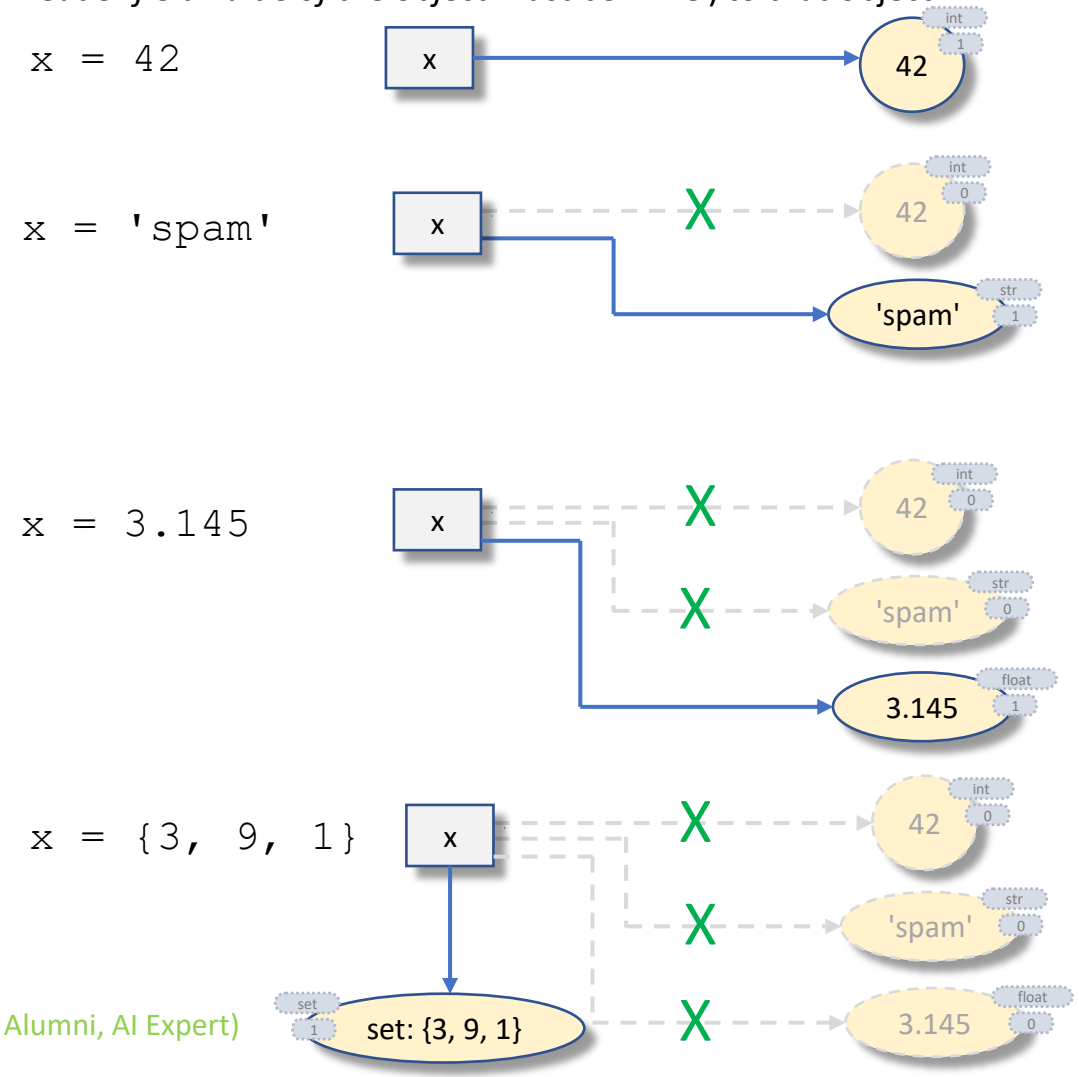


1) Variable Names, Objects are saved in different parts of memory.

2) References are links/pointers connecting those 2 parts of memories

Definition:

Claiming back the object's memory if no references (*reference counter header field value of the object must be ZERO*) to that object.

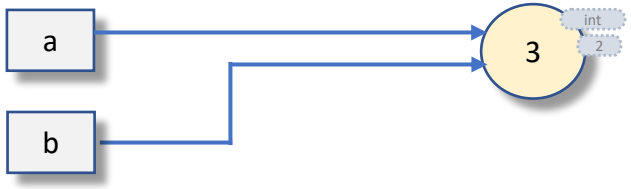


Shared References, Equality

Shared References:

```
a = 3
```

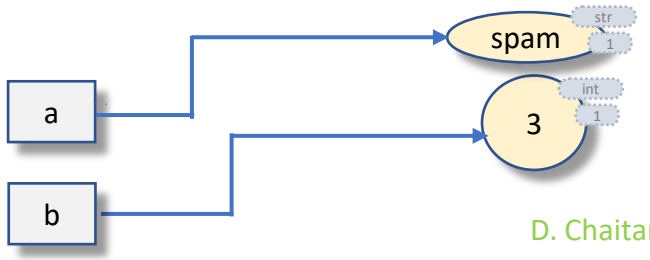

```
b = a
```



Definition:

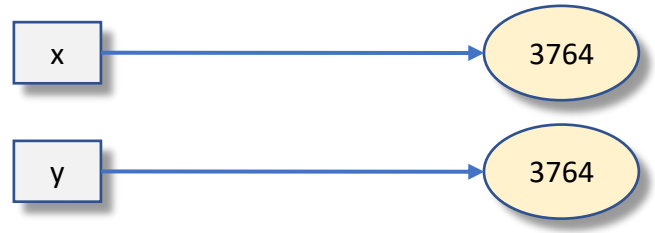
If multiple variables are referring to same object (*reference counter header field value of the object is greater than ONE*), then they shall be called as shared reference

```
a = 'spam'
```



is operator

```
x = 3764
y = 3764
```

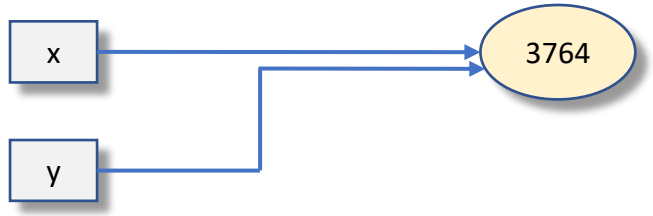



```
x is y   prints False
x == y   prints True
```

object equality check

value equality check

```
x = 3764
y = x
```




```
x is y   prints True
x == y   prints True
```

References *vs* Copies

```
X = [1, 2, 3]
L = ['a', X, 'b']
```

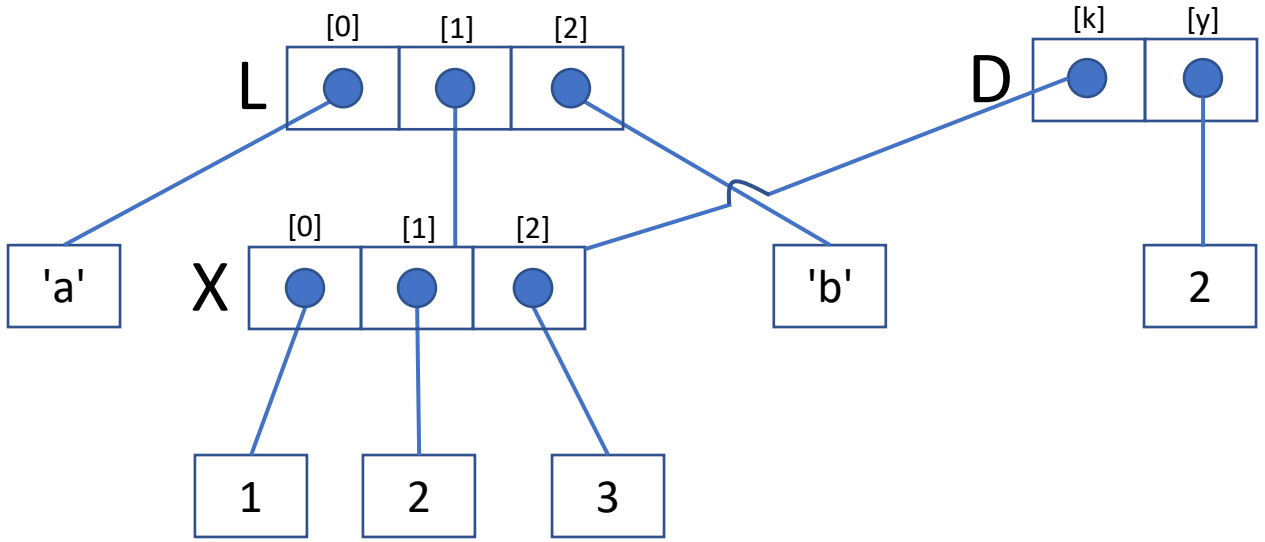
References

```
print(L)
['a', [1, 2, 3], 'b']
```

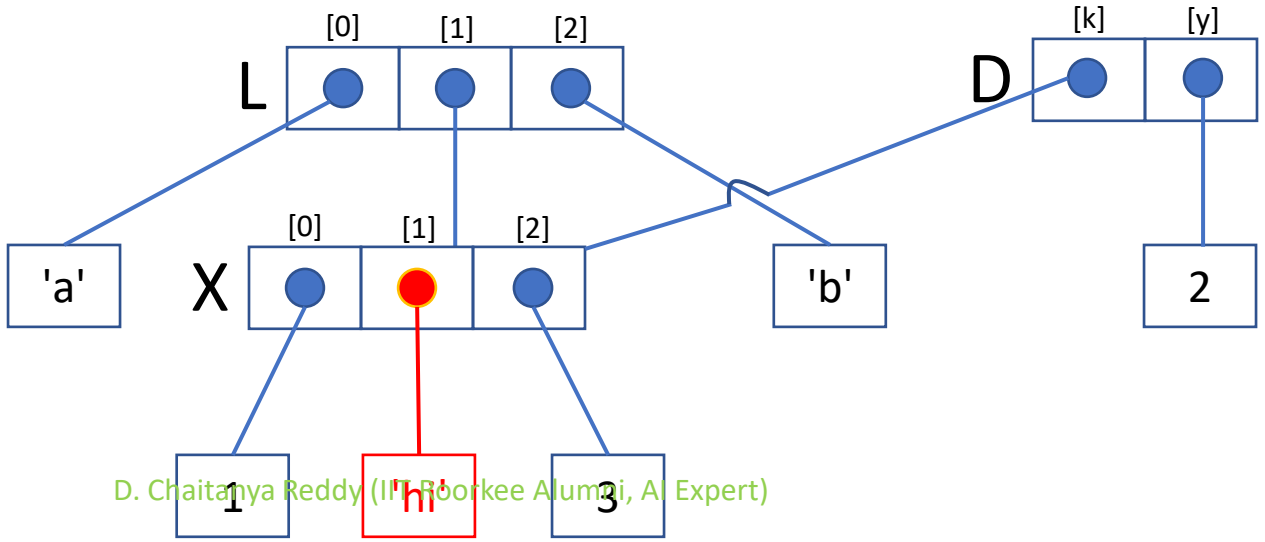
```
D = {'k': X, 'y': 2}
```

References

```
print(D)
{'k': [1, 2, 3], 'y': 2}
```




```
X[1] = 'hi'
print(L)
['a', [1, 'hi', 3], 'b']

print(D)
{'k': [1, 'hi', 3], 'y': 2}
```

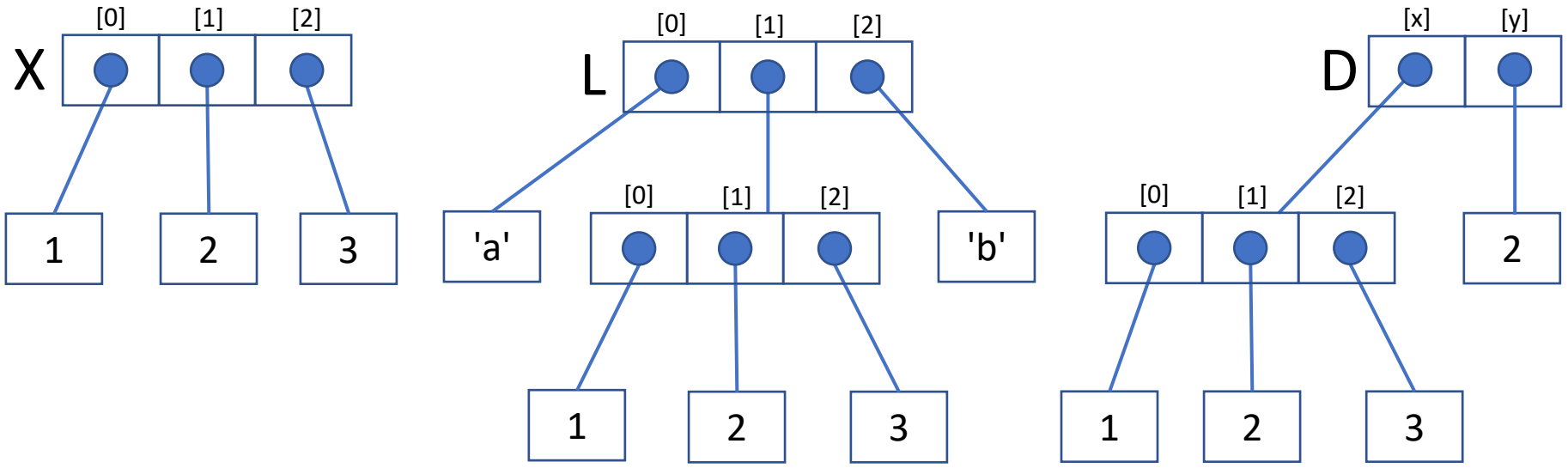


References *vs* Copies

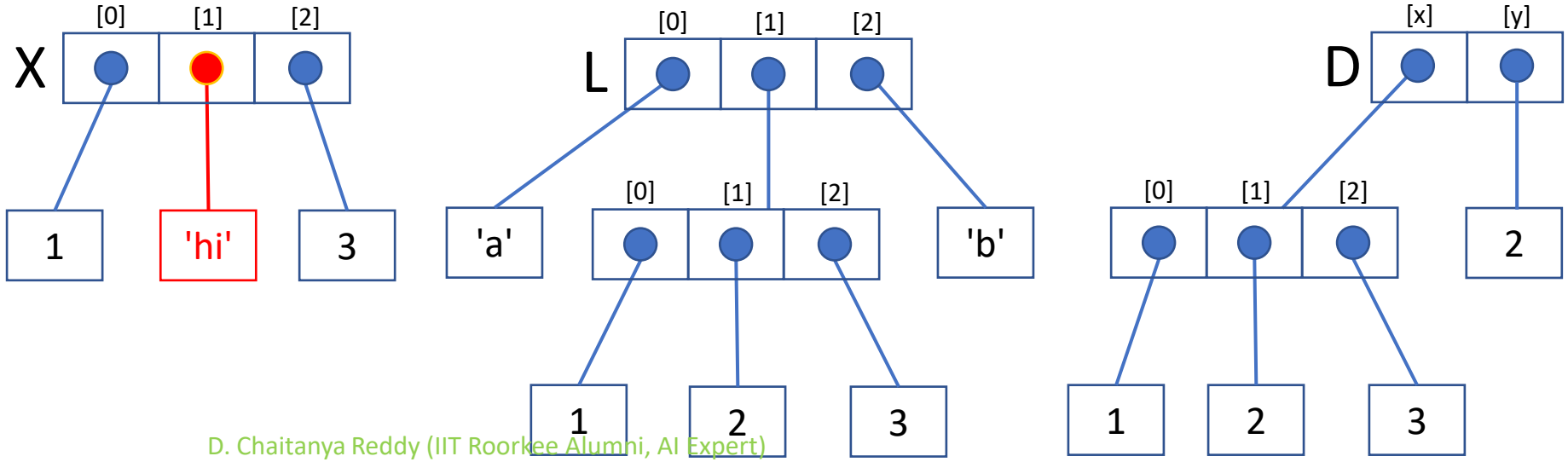
```
X = [1, 2, 3]
L = ['a', X[:], 'b']
D = {'x': X.copy(), 'y': 2}
```

Copies

```
print(L)
['a', [1, 2, 3], 'b']

print(D)
{'x': [1, 2, 3], 'y': 2}
```




```
X[1] = 'hi'
print(X)
[1, 'hi', 3]

print(L)
['a', [1, 2, 3], 'b']

print(D)
{'x': [1, 2, 3], 'y': 2}
```



Statements, Expressions & Syntax

Program Hierarchy

- Programs are composed of modules
- Modules contains statements
- **Statements contains expressions**
- **Expressions create and process objects**

Simple statements

Ex:

```
x = 1
name = 'Mahesh'
print(name)
```

Compound statements

Ex:

```
if x > 100:
    if name == 'python':
        print(name)
else:
    print(x)
```

Python Statements

Name	Purpose / Meaning	Example
Assignment	Creating references	`a = 'good'` `x = 42`
Calls and other expressions	Running functions	`log.write("Hello World")`
print calls	Printing objects	`print('The killer joke')`
If/elif/else	Selecting actions	`if 'python' in text:` `  print(text)`
for/else	Iteration	`for el in mylist:` `  print(el)`
while/else	General loops	`while x > y:` `  print('hello')`
pass	Empty place holder	`while True:` `  pass`
break	Loop exit	`while True:` `  if exittest(): break`
continue	Loop continue	`while True:` `  if skiptest(): continue`
import	Module access	`import sys`
from	Attribute access	`from sys import stdin`
def	Functions and methods	`def myMethod(a, b, c):` `  print(a+b+c)`

Python Statements

Name	Purpose / Role	Example
return	Functions results	`def newMethod(a, b, c):` `  return a+b+c`
yield	Generator functions	`def myGen(n):` `  for i in range(n): yield i*2`

global	Namespaces	`X = 'old'`
		`def function():`
		`  global X,Y`
		`  X = 'new'`

nonlocal	Namespaces	`def outer():`
		`  x = 'old'`
		`  def inner():`
		`  nonlocal x; x = 'new'`

class	Building objects	`class subclass(superclass):`
		`  staticdata = []`
		`  def method(self): pass`

try/except/finally	Managing exceptions	`try:`
		`  action()`
		`except:`
		`  print('action error')`

raise	Triggering exceptions	`raise myException('more details about error')`
assert	Debugging checks	`assert x > y, 'x too small'`
with/as	Context managers	`with open('data.txt') as myfile:` `  process(myfile)`
del	Deleting references	`del data[k] / del data[i:j] / del obj.attr / del a`

Assignment & Expression Statements

Assignment statement forms:

Operation	Interpretation
`s1 = 'spam'`	Basic form
`s1, s2 = 'spam', 'ham'`	Tuple assignment (positional)
`s1 += 42`	Augmented assignment
`[s1, s2] = ['spam', 'ham']`	List assignment (positional)
`a, b, c, d = 'spam'`	Sequence assignment (generalized)
`a, *b = 'spam'`	Extended sequence unpacking
`s1 = s2 = 'hello'`	Multiple target assignment

Both `s1` & `s2` objects referring to same string object (`'hello'`) in memory.

Advantages:

- Less typing.
- Faster in execution.
- Perform in place changes for mutable objects.

Augmented assignment statements

```
X += Y    X &= Y    X -= Y    X |= Y
X *= Y    X ^= Y    X /= Y    X >>= Y
X %= Y    X <<= Y   X **= Y   X //= Y
```

Common Python expression statements

Operation	Interpretation
`spam(eggs, ham)`	Function calls
`spam.ham(eggs)`	Method calls
`spam`	Printing variables in the interactive interpreter
`print(a, b, c, sep='')`	Printing operations
`yield x ** 2`	Yielding expression statements

Variable Name Rules & Conventions

Rules

Must starts with

Must followed by

- Syntax: *(Underscore or letter) + (any number of letters, digits or underscores)*

  Ex: `_spam`, `spam`, `Spam_1` ✓
  `1_spam`, `spam$`, `@#!` ✗

- Case sensitive.

  Ex: `SPam` is not same as `spam`

- Reserved words are off-limit.

`self` is NOT a reserved word in python.

Python 3.X reserved words

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

Conventions

- Begin with underscore (`_X`) shall not imported by a **`from module import *`** statement.
- Have 2 leading and trailing underscores (**`__X__`**) are system defined names that have special meaning to interpreter.
- Begin with 2 underscores and do not end with 2 more (**`__X`**) are localized (mangled) to enclosing classes.
- Just single underscore (**`_`**) retains the result of last expression when working interactively.
- Class names starts with uppercase letter, Module name starts with lower case letter.

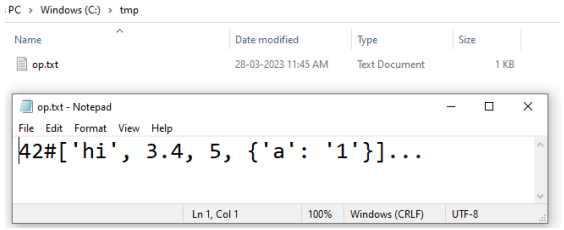
`print` operations

- `print` prints things to the *standard output stream* (called as **`stdout`**).
- Used heavily while debugging the code.
- Syntax:

```
print(object1, object2, ……[, sep='  '][, end='\n'][, file=sys.stdout][, flush=False])
```

Ex:

```
x = 42
my_list = ['hi', 3.4, 5, {'a': '1'}]
print(x, my_list)                          prints   42 ['hi', 3.4, 5, {'a': '1'}]
print(x, my_list, sep='\t')                prints   42      ['hi', 3.4, 5, {'a': '1'}]
print(x, my_list, sep='@', end='!!\n')     prints   42@['hi', 3.4, 5, {'a': '1'}]!!
print(x, my_list, sep='#', end='…', file=open(r'C:\tmp\op.txt', 'w'))
print(x, my_list, sep='$', flush=true)     prints   42$['hi', 3.4, 5, {'a': '1'}]
print()                                    prints   empty line
name, age, weight = 'Mahesh', 45, 72.67
print('%s, age %i yrs, weight %0.2f kg' % (name, age, weight))     prints   Mahesh, age 45 yrs, weight 75.34 kg
print('{}, age {} yrs weight {} kg'.format(name, age, weight))     prints   Mahesh, age 45 yrs, weight 75.34 kg
print(x)                                   prints   42
print(my_list)                             prints   ['hi', 3.4, 5, {'a': '1'}]
```



if-elif-else

if statement:

syntax

```
if SOME_CONDITION:
    single or multi line code
```

Uses multiple variables to form a condition with operator.

1. Should evaluates to True/False.
2. Can use logical **and**, logical **or** for combination of multiple conditions
3. () – parenthesis are optional.

Operator	Meaning
`x < y, x <= y, x > y, x >= y`	Magnitude comparison
`x == y, x != y`	Value equality operators

Uses just variable(s) as a condition.

```
if SOME_VARIABLE:
    single or multi line code
```

1. Should evaluates to True/False.
2. Can use logical **and**, logical **or** for combination of multiple conditions.
3. () – parenthesis are optional.

Object	Interpretation	Value
`'spam'`	Non empty string	True
`''`	Empty string	False
`[1, 2]`	Non empty list or some data structure	True
`[]`	Empty list or some data structure	False
`{'a':1}`	Non empty dictionary	True
`{}`	Empty dictionary	False
`1`	Non zero number	True
`0.0`	Zero number	False
`None`	Not an object. Empty place holder	False

Ex:

```
if x>y:
    x = 1
    y = 2
```

Indentation: inside block

```
if age:
    x = 1
```

Ex:

```
if x>100 and y:
    print(name)
```

if-elif-else

if-else statement:

syntax

```
if SOME_CONDITION:
    single or multi line code
else:
    single or multi line code
```

```
if SOME_VARIABLE:
    single or multi line code
else:
    single or multi line code
```

Can be zero/single `else` block

Ex:

```
if x or y<6:
    print(name)
else:
    print(x)
```

None:

- An object and allocated a memory by python.

Ex:

```
k = None
my_list = [None] * 10
print(my_list)
```

```
[None, None, None, None, None, None, None, None, None, None]
```

if-elif-else

if-elif-else statement:

syntax

```
if SOME_CONDITION:
    single or multi line code
elif SOME_CONDITION:
    single or multi line code
else:
    single or multi line code
```

Can be zero/single/multi `elif` blocks

If a single line code present in any block, the below syntax also allowed:

```
if SOME_CONDITION: single line code
elif SOME_CONDITION:
    single or multi line code
else: single line code
```

Ex:

```
if name=='python':
    print('beginning')
elif name=='machine':
    print('season-1')
elif name=='deep':
    print('season-2')
else:
    print('conclusion')
```

Ex:

```
if name=='python': print('beginning')
elif name=='machine':
    print('season-1')
elif name=='deep': print('season-2')
else: print('conclusion')
```

if-elif-else

Nested `if` statement:

```
if x > 100:
    if name == 'python':
        print(name)
        if x > 110:
            print('begining')
```

Nested `if-else` statement:

```
if x > 100:
    if name == 'python':
        print(name)
else:
    if name == 'machine':
        print('begining')
        if name == 'deep':
            print('conclusion')
        else:
            x = 78
```

if-elif-else

Nested `if-elif-else` statement:

```
if x > 100:
    if name=='python':
        print('hi')
    elif name=='machine':
        print('pushpa')
    else:
        print('RRR')
elif x==100:
    print('bahubali')
else:
    if name == 'machine':
        print('begining')
    if name == 'deep':
        print('conclusion')
    else:
        x = 78
        print('Oscar')
```

`if-else` ternary expression:

```
if year<2022:
    movie = 'bahubali'
else:
    movie = 'RRR'
```

```
movie = 'bahubali' if year<2022 else 'RRR'
```

Syntax:

```
var = VALUE1 if SOME_CONDITION else VALUE2
```

while loop

1. Should evaluates to True/False.
2. Can use logical **and**, logical **or** for combination of multiple conditions.
3. () – parenthesis are optional.

Syntax:

```
while SOME_CONDITION:                #loop test
    single or multi line code        #loop body
else:                                #optinal else
    single or multi line code        #run if and only if loop exit normally.
```

Ex1:

```
a, b = 0, 10
while a < b:
    print(a, end=' ')
    a += 1
```

```
Output: 0 1 2 3 4 5 6 7 8 9
```

Ex2:

```
x = 'spam'
while x:
    print(x, end=' ')
    x = x[1:]
```

```
Output: spam pam am m
```

Ex3:

```
while True:
    print('type Ctrl+C to stop me.')
```

```
Output: type Ctrl+C to stop me.
        .
        .
        .
```

break, continue, pass, loop else

- `break` – jumps out of the closest enclosing loop (skip the entire loop statements).

```
while True:
    name = input('Enter name: ')
    if name=='stop': break
    else:
        age = input('Enter age:')
        print('Hello', name, int(age)*2)
```

```
Output:
Enter name: bob
Enter age: 20
Hello bob 40
Enter name: stop
```

- `continue` – jumps to the start of the closest enclosing loop (to the loop's first line)

```
x = 10
while x:
    x -= 1
    if x%2!=0: continue
    print(x, end=' ')
```

```
Output:
8 6 4 2 0
```

- `pass` – does nothing at all. It's an empty statement placeholder.

```
x = 10
while x:
    pass
```

```
Output:
Infinite loop
```

- `Loop else` – runs if and only if loop is exited normally.

```
x = y // 2
while x > 1:
    if y % x == 0:
        print(y, 'has factor', x)
        break
    x -= 1
else:
    print(y, 'is prime')
```

```
Output:
Prints whether the
y is prime or not.
```

for loop

1. Any object that support iteration protocol or indexing protocol.
2. Loop iterates for every element in object irrespective of any condition.

Syntax:

```
for var in OBJECT:                  #assign OBJECT elements to var
    single or multi line code       #loop body
else:                               #optinal else
    single or multi line code       #run if and only if loop exit normally.
```

Like **while** loop, **break**, **continue**, **pass**, loop **else** are also applicable as it is to **for** loop too.

Ex1:

```
for x in ['spam', 'eggs', 'ham']:
    print(x, end=' ')
```

```
Output: spam eggs ham
```

Ex2:

```
prod = 1
for num in [1, 2, 3, 4, 5]: prod *= num
print(prod)
```

```
Output: 120
```

Ex3:

```
movie = 'bahubali'
for ch in movie: print(ch, end=' ')
```

```
Output: b a h u b a l i
```

Ex4:

```
my_tup = ('I am', 2, 'times', 'good')
for el in my_tup: print(el, end=' ')
```

```
Output: I am 2 times good
```

Ex5:

```
my_list = [(1, 2), (7, 14), (-2, -4)]
for (a, b) in my_list: print(a, b, sep='x2= ')
```

```
Output:
1x2= 2
7x2= 14
-2x2= -4
```

for loop

Ex6:

```
L1 = [(1, 2, 3), (7, 14, 21), (-2, -4, -6)]
for (a, b, c) in L1: print(a, b, c, sep='>>')
```

```
Output:
1 >> 2 >> 3
7 >> 14 >> 21
-2 >> -4 >> -6
```

Ex7:

```
L2 = [((1, 2), 3), ((7, 14), 21), ((-2, -4), -6)]
for ((a, b), c) in L2: print(a, b, c, end=', ')
```

```
Output:
1 2 3, 7 14 21, -2 -4 -6,
```

Ex8:

```
L3 = [(1, 2, 3, 4), (10, 20, 30, 40)]
for T in L3:
        print(T[-1], T[1:], T[0:2], end=', ')
```

```
Output:
4 (2, 3, 4) (1, 2), 40 (20, 30, 40) (10, 20),
```

Ex9:

```
D1 = {'a': 2, 'b': 4, 'c': 16}
for k in D1:
   print(k, D1[k], sep='=>', end=', ')

for k in list(D1.keys()):
   print(k, D1[k], sep='=>', end=', ')

for (k, vl) in list(D1.items()):
   print(k, vl, sep='=>', end=', ')
```

```
Output: a=>2, b=>4, c=>16
```

Ex10:

```
for line in open('c:\tmp\info.txt').readlines():
   print(line)

for line in open('c:\tmp\info.txt', 'r'):
   print(line)
```

```
Output: prints line by line the content
in info.txt file.
```

Nested `for` loop

A `for` loop inside another `for` loop inside another `for` loop soon…

Ex:

```
items = ['aaa', 111, (4, 5), 2.01]
tests = [(4, 5), 3.14]
for key in tests:
    for item in tests:
        if item==key:
            print(key, 'was found')
            break
    else:
        print(key, 'not found!')
```

```
Output:
(4, 5) was found
3.14 not found!
```

1. Like **`for`** `loop`, **nesting** also applicable as it is to **`while`** loop too.
2. All the examples shown for **`for`** loop can also achieved with **`while`** loop, but needs more lines of code.
3. for loop has many benefits (comprehensions, manual indexing, etc…) compared to while loop in python.

Built-in functions used with `for`

`range`: built-in function produces a series of successively higher integers, which can be used as indexes in a `for`

```
>>> range(5)
range(0, 5)
>>> list(range(5))
[0, 1, 2, 3, 4]
>>>list(range(2, 5))
[2, 3, 4]
>>>list(range(0, 10, 2))
[0, 2, 4, 6, 8]
>>>list(range(-3, 3))
[-3, -2, -1, 0, 1, 2, 3]
>>> list(range(4, -5, -2))
[4, 2, 0, -2, -4]
```

```
for i in range(3):
    print(i**3, '% success', end=', ')
```

No need to write `list(range(3))` as range() support indexing protocol.

```
Output: 0 % success, 1 % success, 8 % success,
```

```
X = 'spam'
for ix in range(len(X)):
    X = X[ix:] + X[:ix]
    print(X, end=' ')
```

```
Output: spam, pams mspa amsp
```

```
S = 'abcdefghijk'
for id in range(0, len(S), 2):
    print(S[id], end=' ')
```

```
Output: a c e g i k
```

```
S = 'abcdefghijk'
for ch in S[0:len(S):2]:
    print(S[id], end=' ')
```

```
Output: a c e g i k
```

```
S = 'abcdefghijk'
for ch in S[::2]:
    print(S[id], end=' ')
```

```
Output: a c e g i k
```

Same output, but multiple ways to write with `for`

Built-in functions used with `for`

`zip`: built-in function returns a series of parallel-item tuples which can be used to traverse multiple sequences in a `for`

```
>>> L=[1, 2, 3, 4]
>>> LS=[1, 4, 9, 16]
>>> LQ=[1, 8, 27, 64]
>>> zip(L, LS)
<zip at 0x1f3356f40c0>
>>>list(zip(L, LS))
[(1, 1), (2, 4), (3, 9), (4, 16)]
>>>list(zip(L, LS, LQ))
[(1, 1, 1), (2, 4, 8), (3, 9, 27), (4, 16, 64)]
for (n, s, q) in zip(L, LS, LQ):
   print(n, s, q, sep='=>')
Output: 1=>1=>1
        2=>4=>8 …
```

No need to write `list(zip(L, LS, LQ))` as zip() is iterable.

```
S1='abc'
S2='xyz123'
for (c1, c2) in zip(S1, S2):
   print(c1, c2, sep='=>', end=', ')
Output: a=>x, b=>y, c=>z,
```

If both sequences are not of same length, zip truncates to shortest length

```
keys=['spam', 'eggs', 'ham']
values=[1, 7, 11]
D = dict()
for (key, val) in zip(keys, values):
   D[key]=val
print(D)
Output: {'eggs': 7, 'ham': 11, 'spam': 1}
```

Built-in functions used with `for`

`map`: built-in function applies a passed-in function to each element in iterable object and returns a list contains all function call results.

```
>>> L=[-3, -6, 0, 9, 14]
>>> map(abs, L)
<map at 0x1f3369d84c0>
>>> list(map(abs, L))
[3, 6, 0, 9, 14]
```

No need to write `list(map(math.sqrt, LN))` as map() is iterable.

```
LN=[1, 4, 34, 56, 75]
for el in map(math.sqrt, LN):
    print(int(el, end=', ')
Output: 1.0, 2.0, 5.83, 7.48, 8.66,
map(math.pow, LN, [2, 3, 0, 1, 0.5])
```

`filter`: a built-in function applies a passed-in function to each element in iterable object and returns a list contains all elements for which the function call resulted to `True`.

```
>>> L=[-3, -6, 0, 9, 14]
>>> filter(lambda x: x>0, L)
<filter at 0x1c431b97e20>
>>> list(filter(lambda x: x>0, L))
[9, 14]
```

No need to write `list(filter(lamb…, LN))` as filter() is iterable.

```
LN=[1, 4, 34, 56, 75]
for el in filter(lambda x: x>10 and x<65, LN):
    print(el, end=',')
Output: 34,56,
```

Built-in functions used with `for`

`enumerate`: built-in function generates both the values and indexes of the items in an iterable so we do not need count manually.

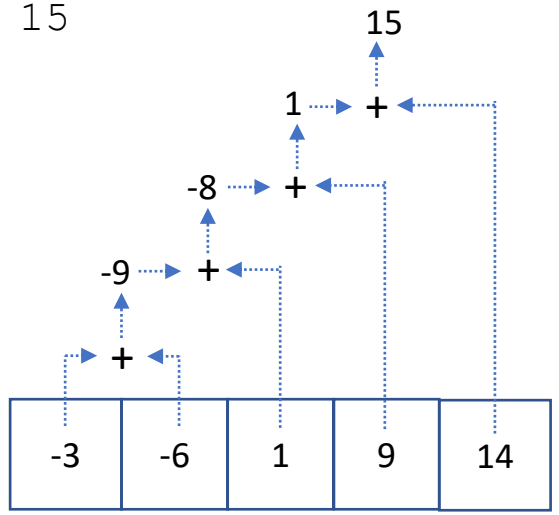
```
>>> S='pushpa'
>>> enumerate(S)
<enumerate at 0x1f33568ff80>
>>>list(enumerate(S))
[(0, 'p'), (1, 'u'), (2, 's'), (3, 'h'), (4, 'p'), (5, 'a')]
```

No need to write `list(enumerate(S))` as enumerate() is iterable.

```
for (i, v) in enumerate(S):
    print(v*i, end=', ')
Output: , u, ss, hhh, pppp, aaaaa,
```

`reduce`: a function present in `functools` module that accepts a function and iterable, will return a single element.

```
>>> L=[-3, -6, 1, 9, 14]
>>> from functools import reduce
    reduce(lambda x, y: x+y, L)
Output: 15
```




```
>>> reduce(lambda x, y: x*y, L)
Output: 2268
```

Iterables & Iterations

Recap….

Syntax:

1. Any object that support iteration protocol or indexing protocol.
2. Loop iterates for every element in object irrespective of any condition.

```
for var in OBJECT:                    #assign OBJECT elements to var
    single or multi line code         #loop body
else:                                 #optinal else
    single or multi line code         #run if and only if loop exit normally.
```

Iteration Protocol: A Protocol that supports `iter()` and `__next__()` calls.

Indexing Protocol: A Protocol that supports `indexing operation like [0], [4].`

```
R=range(-5, 0)
R[1], R[0], R[4]
```

Supporting indexing operation

```
Output:-4, -5, -1
```

Can still create **iterable** object that supports indexing operation

```
I=iter(R)
I.__next__(), I.__next__(), I.__next__()
```

Manual iterations

```
Z = zip([1, 2, 3], [11, 12, 13], [1, 4, 9])
IZ=iter(Z)
```

Create **Iterable** object for `range` object.

```
IZ.__next__(), IZ.__next__(), IZ.__next__()
(1, 11, 1), (2, 12, 4), (3, 13, 9)
```

Manual iterations

```
F=open(r'C:\tmp\info.txt')            #call iter() directly
F.__next__(), F.__next__(), F.__next__()
1st line of file, 2nd line of file, 3rd line of file
```

Calling `__next__()` method raises a built-in exception: `StopIteration` after the last element in iterable object.

```
M = map(math.pow, [1, 2, 3], [2, 3, 3])
IM = iter(M)
IM.__next__(), IM.__next__(), IM.__next__()
(1.0, 8.0, 27.0)
IM.__next__()
Raises StopIteration error
```

Protocols Summary

Indexing Protocol

- Protocol that supports indexing operation.
- Elements can be accessed using notation `obj[0]`, `obj[1]`, etc…
- Can access element at any index at any time.
- Elements in object can be accessed multiple times without reinitializing object.
- Accessing elements with wrong index results into `IndexError`.

Iteration Protocol

- Protocol that supports `iter()` or `__iter__()` and `__next__()` calls.
- Elements can be accessed using `__next__()` method call.
- Should start accessing the elements from the start and access sequentially only using `__next__()` call.
- Elements in object can be accessed only once. Need to reinitialize the object to access elements again.
- Calling __next__() after fetching the last element results into `StopIteration` Error.

Comprehensions

1) Less coding
2) Run much faster.

List with `for` loop:

```
L = list(range(0, 20))
LN = []
for x in L:
    val = x/2
    LN.append(val)
print(LN)
```

```
Output: [0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 5.5, 6.0, 6.5, 7.0, 7.5, 8.0, 8.5, 9.0, 9.5]
```

List Comprehension:

```
print([x/2 for x in range(0, 20)])
```

Set Comprehension:

```
print({int(math.pow(x, 3)/math.sqrt(x)) for x in range(1, 20)})
```

```
Output: {1024, 1, 129, 5, 15, 401, 32, 1573, 1191, 181, 55, 316, 88, 733, 1374, 609, 871, 498, 243}
```

Dictionary Comprehension:

```
print({x: int(math.pow(x, 3)/math.sqrt(x)) for x in range(1, 20)})
```

```
Output: {1: 1, 2: 5, 3: 15, 4: 32, 5: 55, 6: 88, 7: 129, 8: 181, 9: 243, 10: 316, 11: 401, 12: 498, 13: 609,
14: 733, 15: 871, 16: 1024, 17: 1191, 18: 1374, 19: 1573}
```

With files:

```
[('2023' in line, line[:5]) for line in open(r'C:\tmp\test.txt')]
```

```
Output: [(False, 'bahub'), (True, 'RRR-2'), (False, 'Pushp'), (False, 'Avata')]
```

bahubali-2016
RRR-2023
Pushpa-2022
Avatar-2021

Comprehensions

List with `for` loop with `if` condition:

```
L = list(range(0, 20))
LN = []
for x in L:
    if x%2==0:
        val = x/2
        LN.append(val)
print(LN)
```

Output: [0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]

List Comprehension with `if` filter:

```
print([x/2 for x in range(0, 20) if x%2==0])
```

Set Comprehension:

```
print({int(math.pow(x, 3)/math.sqrt(x)) for x in range(1, 20) if x%2==0})
```

Output: {32, 1024, 5, 498, 181, 88, 316, 733, 1374}

Dictionary Comprehension:

```
print({x: int(math.pow(x, 3)/math.sqrt(x)) for x in range(1, 20) if x%2==0})
```

Output: {2: 5, 4: 32, 6: 88, 8: 181, 10: 316, 12: 498, 14: 733, 16: 1024, 18: 1374}

With Files:

```
[('2023' in line, line[:5]) for line in open(r'C:\tmp\test.txt') if '???' not in line]
```

Output: [(False, 'bahub'), (True, 'RRR-2'), (False, 'Pushp'), (False, 'Avata')]

bahubali-2016
RRR-2023
Pushpa-2022
Avatar-2021
Pushpa2-???

Comprehensions

Nested loops:

```
res = []
for x in 'abc':
   for y in 'lmn':
      res.append(x+y)

print(res)
```

List Comprehension with nesting:

```
print([x+y for x in 'abc' for y in 'lmn'])
```

```
Output: ['al', 'am', 'an', 'bl', 'bm', 'bn', 'cl', 'cm', 'cn']
```

Nesting with `if` filter:

```
res = []
RL = [1, 0, 0, 2, 3]
L = range(1, 6)
for x in L:
   if x%2==0:
      for y in RL:
         if y>0: res.append(int(math.pow(x, y)/math.sqrt(x)))
print(res)
Output: [1, 2, 5, 2, 8, 32]
```

List Comprehension with nesting, `if` filter:

```
print([int(math.pow(x, y)/math.sqrt(x)) for x in range(1, 6) if x%2==0 for y in RL if y>0])
```

Documentation, Help

Option	Role	Example
`dir` function	List of attributes available in objects	`dir([])` `dir(tuple)`

Docstrings:`__doc__`	In-file documentation attached to objects	`import collections`
		`print(collections.__doc__)`
		`print(list.__doc__)`

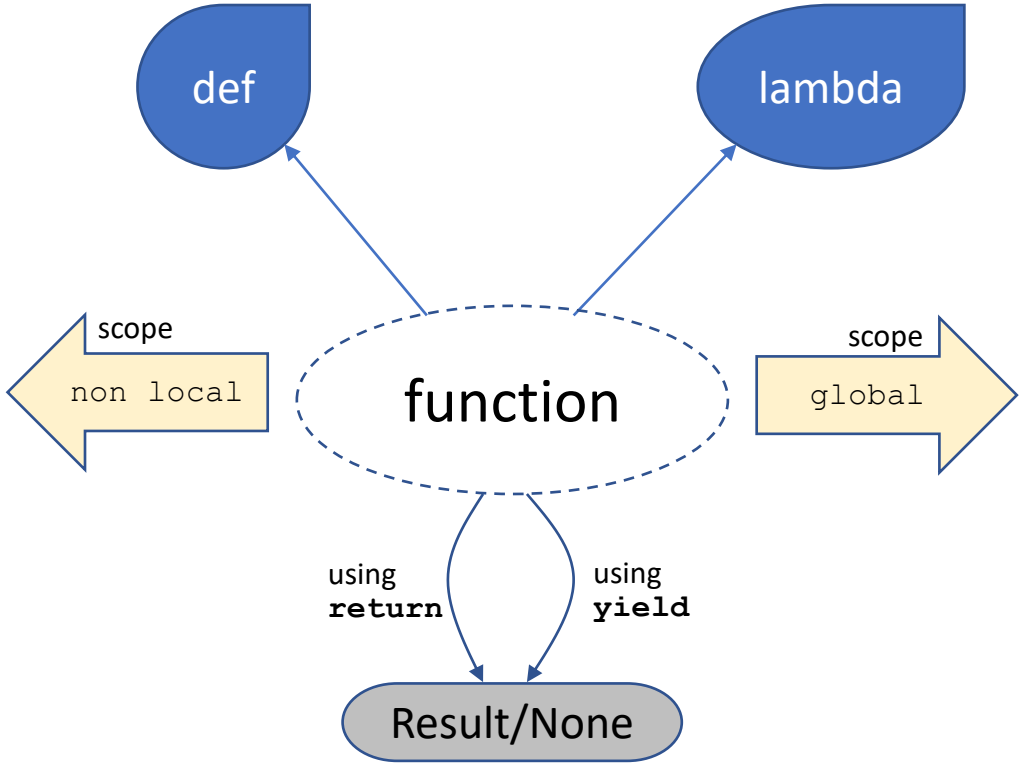
`help` function	Interactive help for objects	`help(list)`
		`import sys`
		`help(sys)`

`PyDoc`: HTML reports	Module documentation in a browser	`python –m pydoc –b` (in cmd)

Functions, Scopes & Arguments

Functions

- A set of code statements so that it can called/run multiple times in a program.
- Optionally return result back to callers.
- Can be nested too.
- Most basic program structure python provides to maximize the code reuse.
- Helps us to design complex programs
- Allow us to split larger systems into manageable parts.



Statement or expression	Examples
Call expressions	`myfunc('spam', 'eggs', meat=ham, *rest)`
def	`def printer(messge):` `  print('Hello ' + message)`
return	`def adder(a, b=1, *c):` `  return a + b + c[0]`

Statement or expression	Examples

global	`x = 'old'`
	`def changer():`
	`  global x; x = 'new'`

nonlocal (3.X)	`def outer():`
	`  x = 'old'`
	`  def changer():`
	`  nonlocal x; x = 'new'`

yield	`def squares(x):` `  for i in range(x): yield i ** 2`
lambda	`funcs = [lambda x: x**2, lambda x: x**3]`

Functions

Syntax:

```
def name(arg1, arg2, ……, argN):
    ...
    ...
    return value
```

1. Returning value is **optional**.
2. If NO **return** means, return **None**
3. Just **return** means, return **None**

- defs can be nested

```
X = 99

def f1():
    name = 'Ganesh'
    def f2():
        print('Hello World!')
    f2()

f1()
```

- defs are not evaluated until they are reached to run.
- defs are not reached to run until they are called.

- defs are also nothing but objects.

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

sq_val = get_square(4)

my_func = get_square
result = my_func(5)
```

- Can attach arbitrary attributes to defs to store some information.

```
get_square.last_num = 5
get_square.list_nums = []
get_square.list_nums.append(4)
get_square.list_nums.append(5)
```

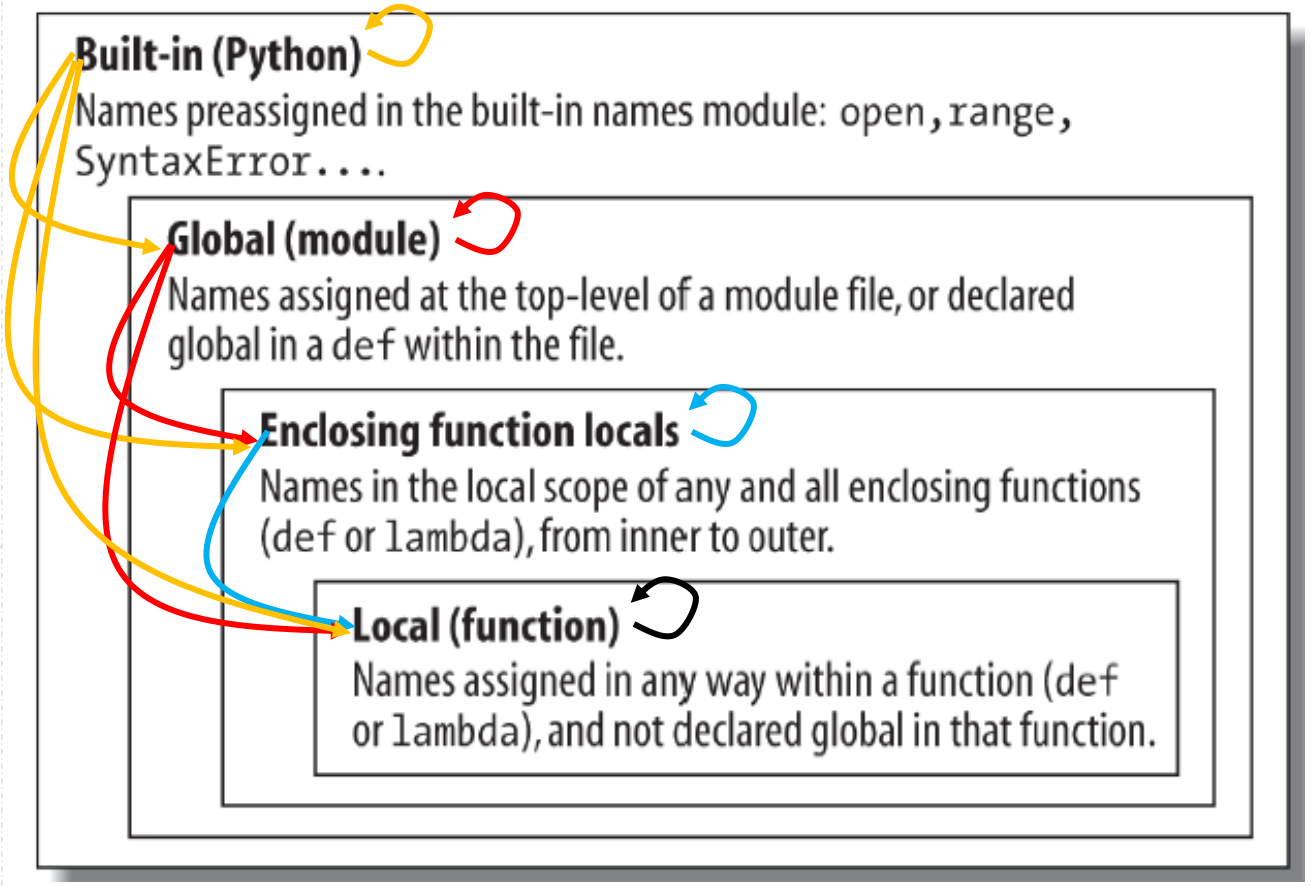
Scopes

The scope of variable is determined only by location of variables created in source code of your program files.

```
name = 'Mahesh'                 # Global (module) scope name

def func():
    name = 'Suresh'             # Local(function) scope name : a different variable
    print(name)

print(name)

func()
```

```
name = 'Mahesh'                 # Global (module) scope name

def bunc():
    info = 'Suresh'             # Local(function) scope name : a different variable
    def junc():
        age = 56
        print(age)
    print(info)
    junc()
print(name)
bunc()
```

L E G B rule



Scopes

Example 1: No name collisions, Hiding

```
X = 99

def f1():
    name = 'Ganesh'
    def f2():
        weight = 67.4
        print('Hello World!')
        print(X, name, weight)
    f2()

f1()
```

```
Output:
Hello World!
99 Ganesh 67.4
```

Example 2: Names hided

```
X = 99

def f1():
    name, X = 'Ganesh', 88
    print('pushpa', X, name)
    def f2():
        weight, name, X = 67.4, 'Suresh', 77
        print('Hello World!')
        print(X, name, weight)
    f2()

f1()
print(X)
```

```
Output:
Pushpa 88 Ganesh
Hello World!
77 Suresh 67.4
99
```

Example 3: Do not want to hide names, create global differently

```
X = 99

def f1():
    global X
    global Y
    name, X, Y = 'Ganesh', 88, 0
    print('pushpa', X, name)
    def f2():
        global X
        global Y
        global Z
        nonlocal name
        weight, name, X, Y, Z = 67.4, 'Suresh', 77, 1, 10.99
        print('Hello World!')
        print(X, name, weight)
    f2()
    print('RRR', X, name, Y)

f1()
print(X, Y, Z)
```

```
Output:
pushpa 88 Ganesh
Hello World!
77 Suresh 67.4
RRR 77 Suresh 1
77 1 10.99
```

Accessing globals differently, Factory Functions

#thismod.py

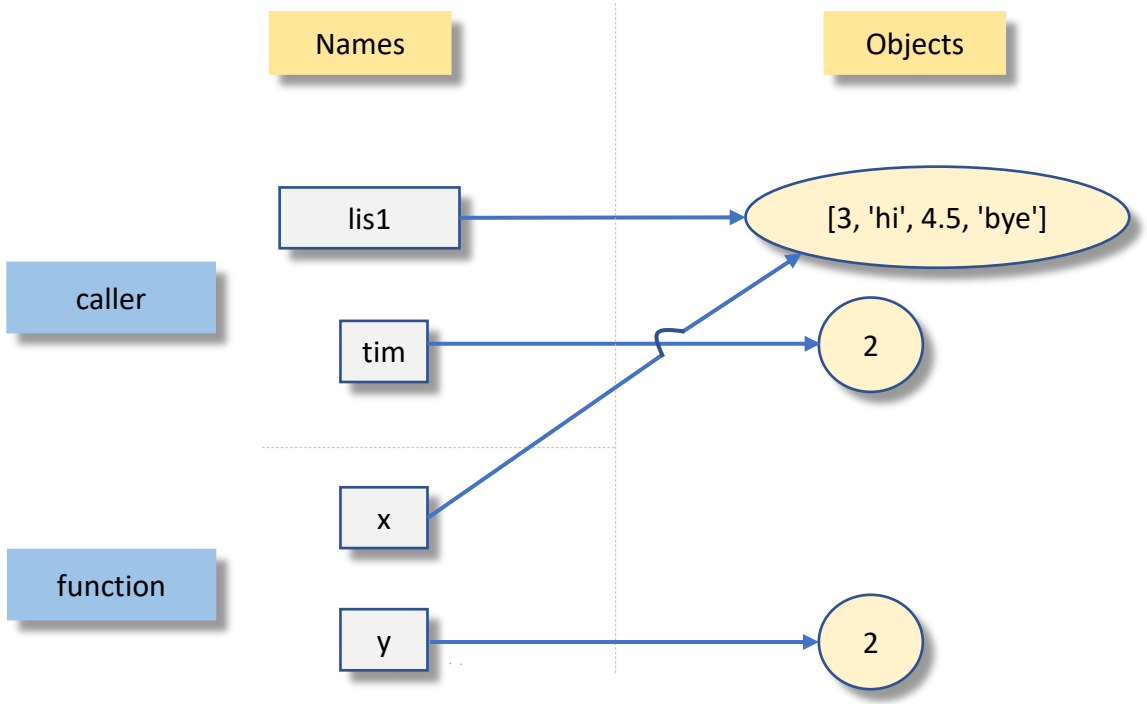
```
var = 99

def local():
    var = 0

def glob1():
    global var
    var += 1

def glob2():
    var = 0
    import thismod
    thismod.var += 1

def glob3():
    var = 0
    import sys
    mod_name = sys.modules['thismod']
    mod_name.var += 1

def test():
    print(var)
    local(); glob1(); glob2(); glob3()
    print(var)

test()
```

```
Output:
99
102
```

```
def maker(N):
    def action(X):
        return X**N
    return action


f = maker(2)
print(f)
Output: <function maker.<locals>.action at 0x00000133177BE7A0>


print(f(3))
Output: 9

print(f(4))
Output: 16

g = maker(3)
print(g(4))
Output: 64

print(f(4))
Output: 16
```

Arguments

- Immutable arguments are passed by value.
- Mutable arguments are passed by reference.



#test1.py

Function definition, arguments

```
def multiply(x, y):
    y += 1
    if isinstance(x, list):
        x.pop(-1)
    elif isinstance(x, str):
        x = x+' - good'
    else:
        x = x+1

    return x*y, x, y

lis1 = [3, 'hi', 4.5, 'bye']
tim = 2
res = multiply(lis1, tim)
print(res, lis1, tim)
Output: ([3, 'hi', 4.5, 3, 'hi', 4.5, 3, 'hi',
4.5], [3, 'hi', 4.5], 2)

s1 = 'Mahesh'
tim = 2
result, s1_re, tim_re = multiply(s1, tim)
print(result, s1_re, tim_re)
Output: Mahesh - goodMahesh - goodMahesh - good
Mahesh - good 3
```

```
x = x.copy(); x.pop(-1)
or
x = x[:-1]
```

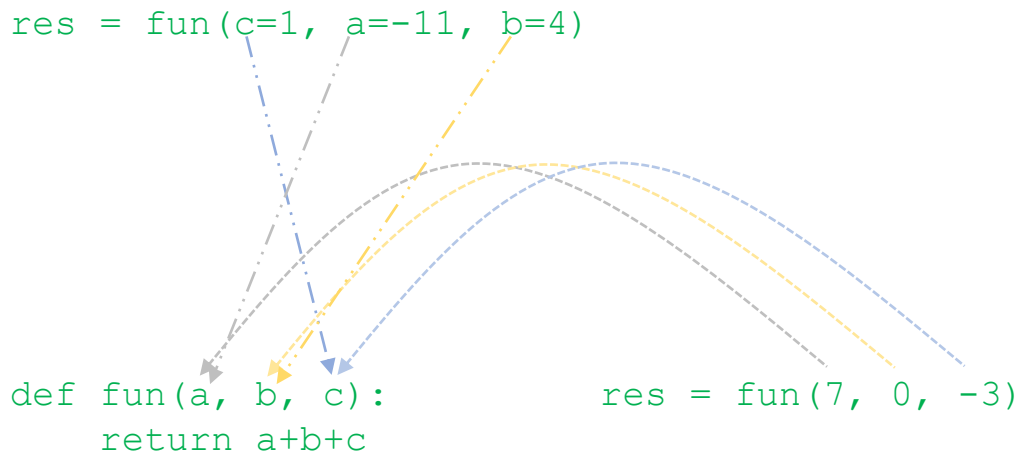
Arguments matching by position

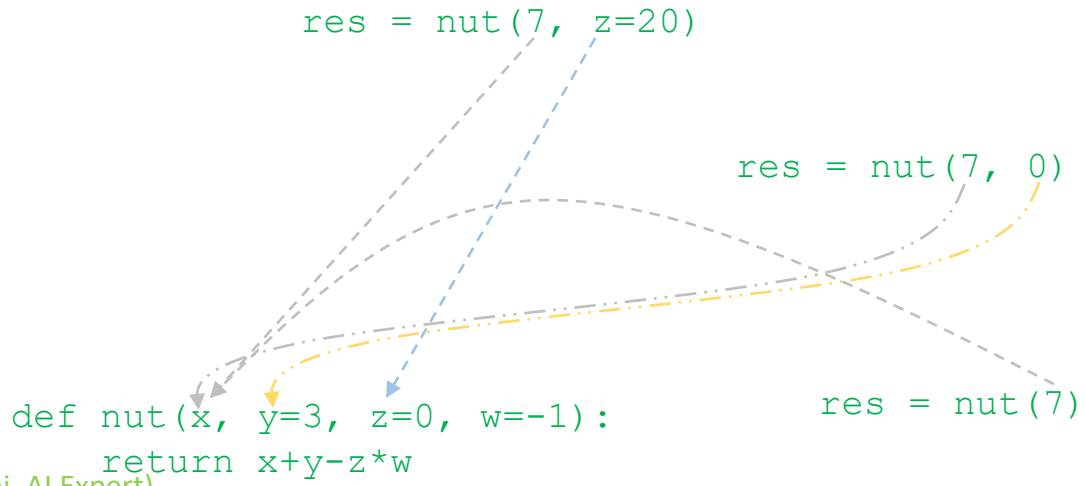
Caller, arguments

Caller, arguments

Matching Args, Passing Args

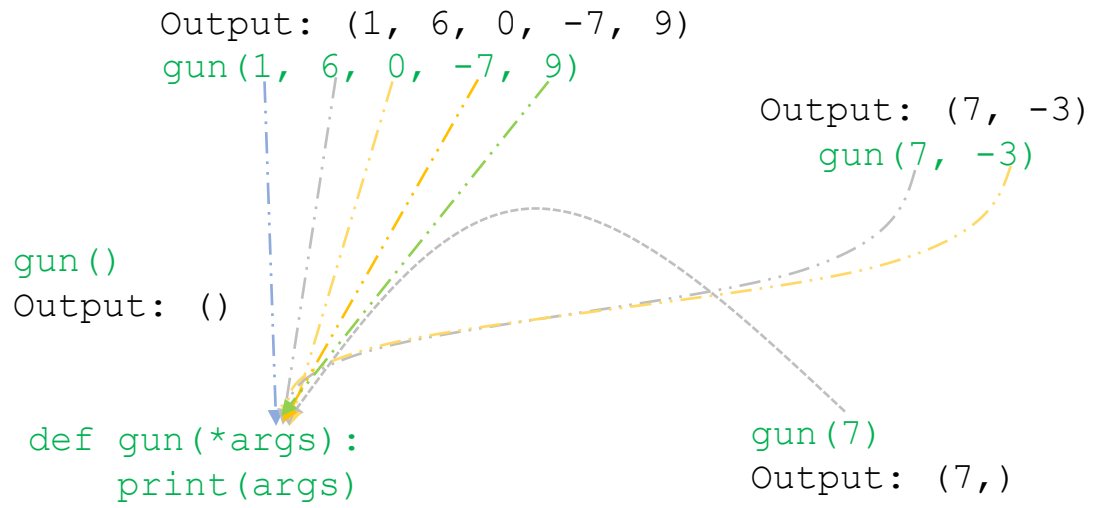
- Matching by position from left to right – positional args
- Matching by argument name – keyword args
- Specify default values for optional args
- Collect arbitrarily many positional or keyword args
- Pass arbitrarily many positional or keyword args
- Args must be passed by name – keyword only args

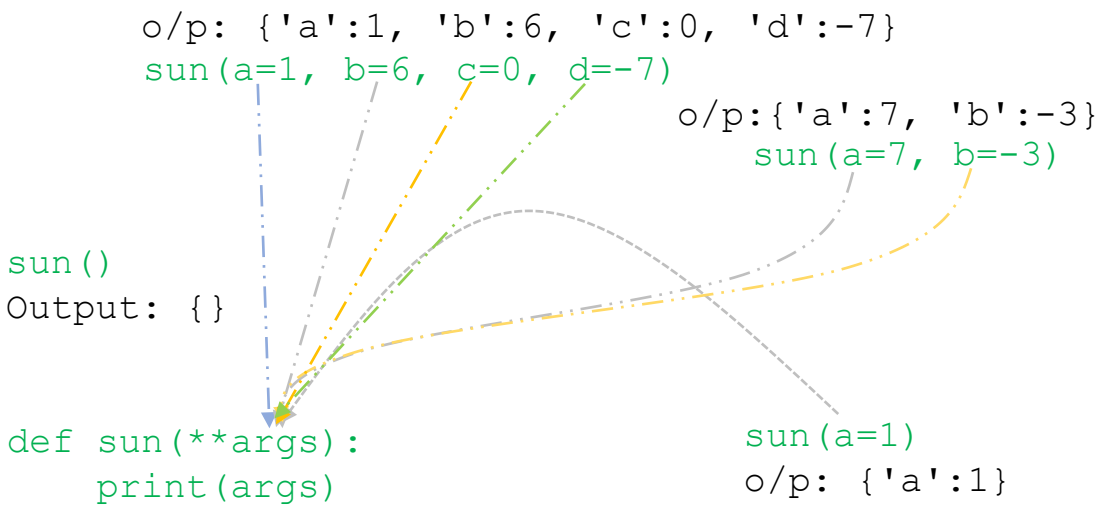




Matching Args, Passing Args

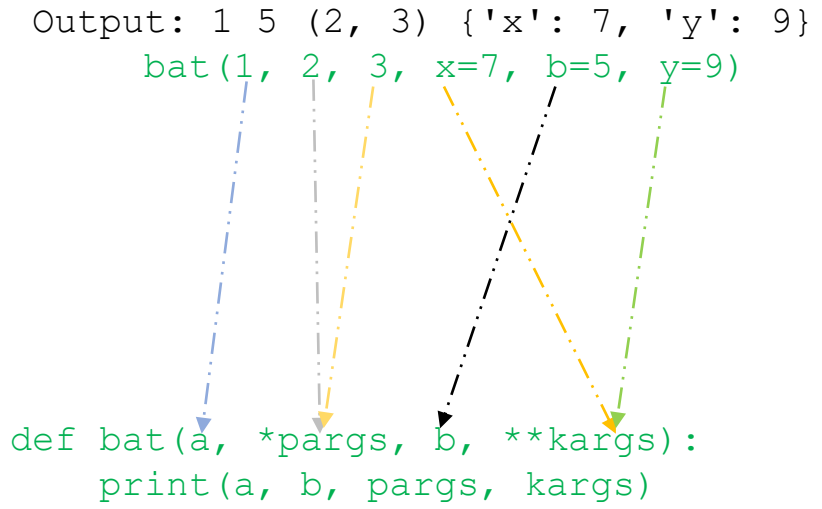
- Matching by position from left to right – positional args
- Matching by argument name – keyword args
- Specify default values for optional args
- Collect arbitrarily many positional or keyword args
- Pass arbitrarily many positional or keyword args
- Args must be passed by name – keyword only args





Matching Args, Passing Args

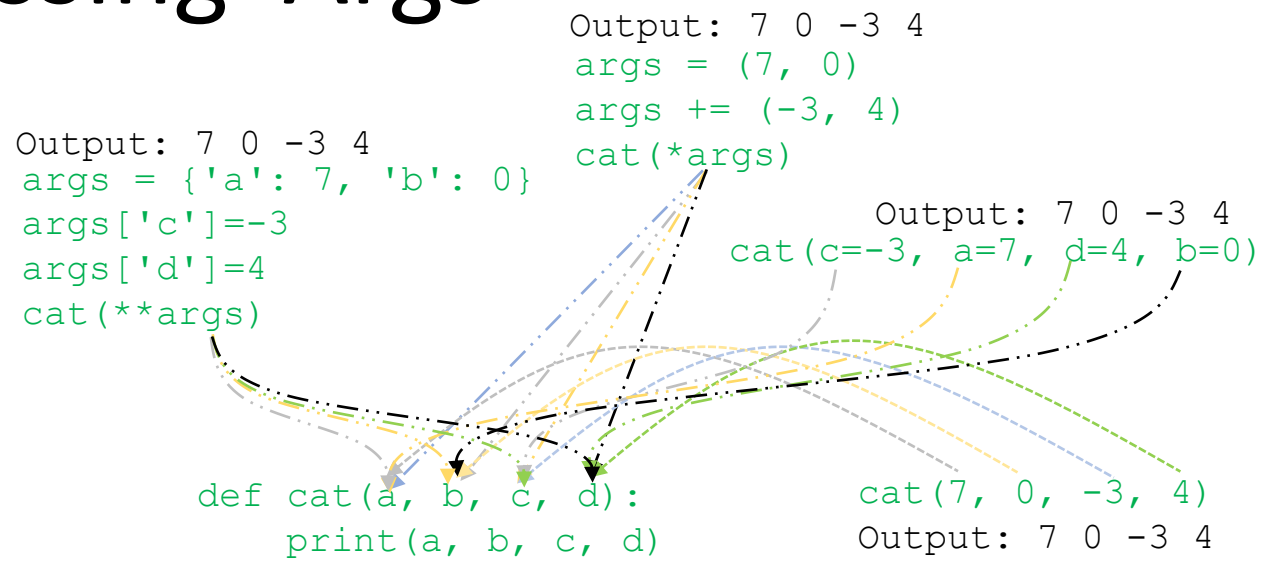
- Matching by position from left to right – positional args
- Matching by argument name – keyword args
- Specify default values for optional args
- Collect arbitrarily many positional or keyword args
- Pass arbitrarily many positional or keyword args
- Args must be passed by name – keyword only args



1. Assign nonkeyword args by matching position
2. Assign keyword args by matching names
3. Assign extra nonkeyword args to `*argname` tuple
4. Assign extra keyword args to `**argname` dictionary
5. Assign default values to unassigned args in header.

Matching Args, Passing Args

- Matching by position from left to right – positional args
- Matching by argument name – keyword args
- Specify default values for optional args
- Collect arbitrarily many positional or keyword args
- Pass arbitrarily many positional or keyword args
- Args must be passed by name – keyword only args

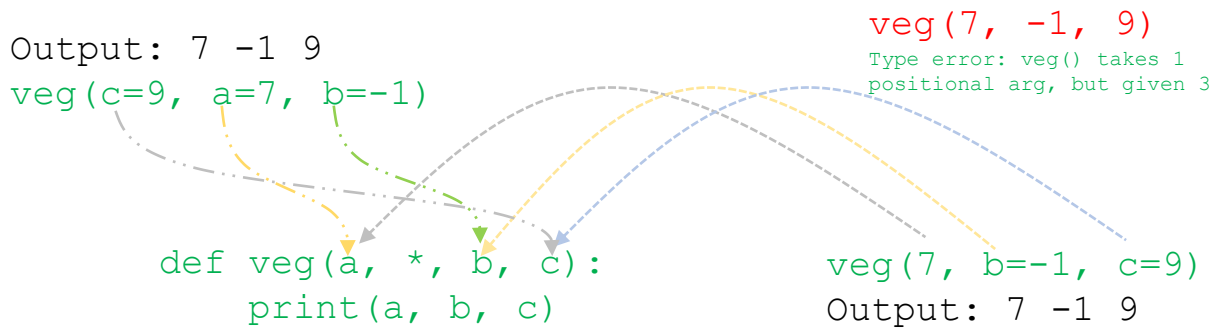


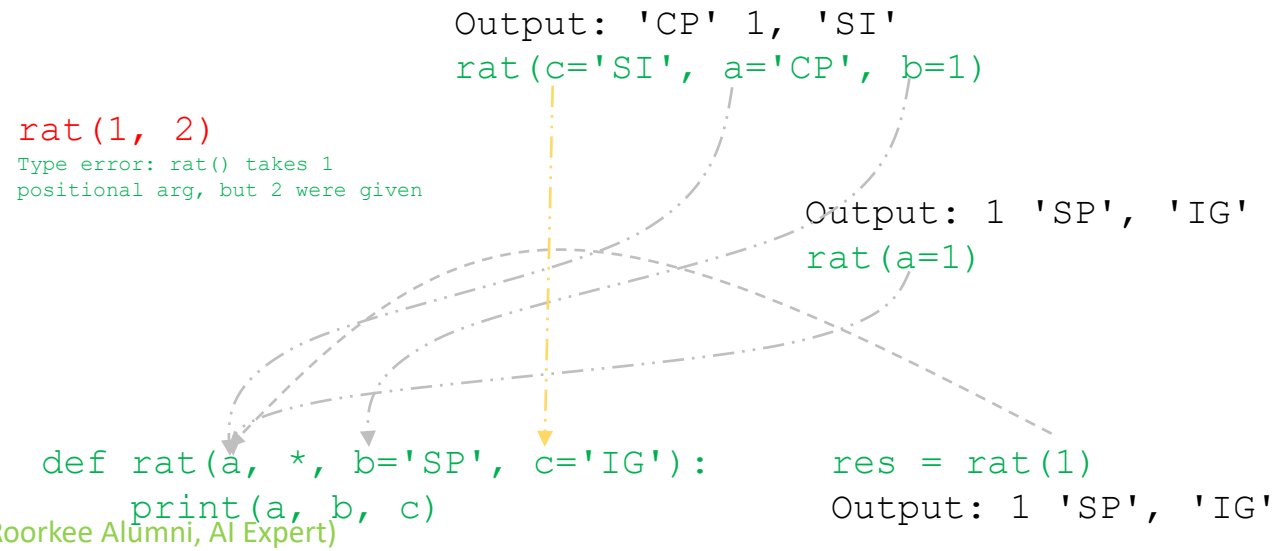

```
cat(*(1, 2), **{'d':4, 'c':3})      # same as calling cat(1, 2, d=4, c=3)
Output: 1 2 3 4

cat(1, *(2, 3), **{'d':4})          # same as calling cat(1, 2, 3, d=4)
Output: 1 2 3 4

cat(1, c=3, *(2,), **{'d':4})       # same as calling cat(1, 2, c=3, d=4)
Output: 1 2 3 4

cat(1, *(2, 3), d=4)                # same as calling cat(1, 2, 3, d=4)
Output: 1 2 3 4

cat(1, *(2,), c=3, **{'d':4})       # same as calling cat(1, 2, d=4, c=3)
Output: 1 2 3 4
```

Matching Args, Passing Args

- Matching by position from left to right – positional args
- Matching by argument name – keyword args
- Specify default values for optional args
- Collect arbitrarily many positional or keyword args
- Pass arbitrarily many positional or keyword args
- Args must be passed by name – keyword only args

1. All args appear after * in function header are **keyword only** args.
2. **keyword only** args must be present before **args and after *args, when both are present in function header.
3. Arg name appearing before *args are default args, not **keyword only** args.





Matching Args, Passing Args

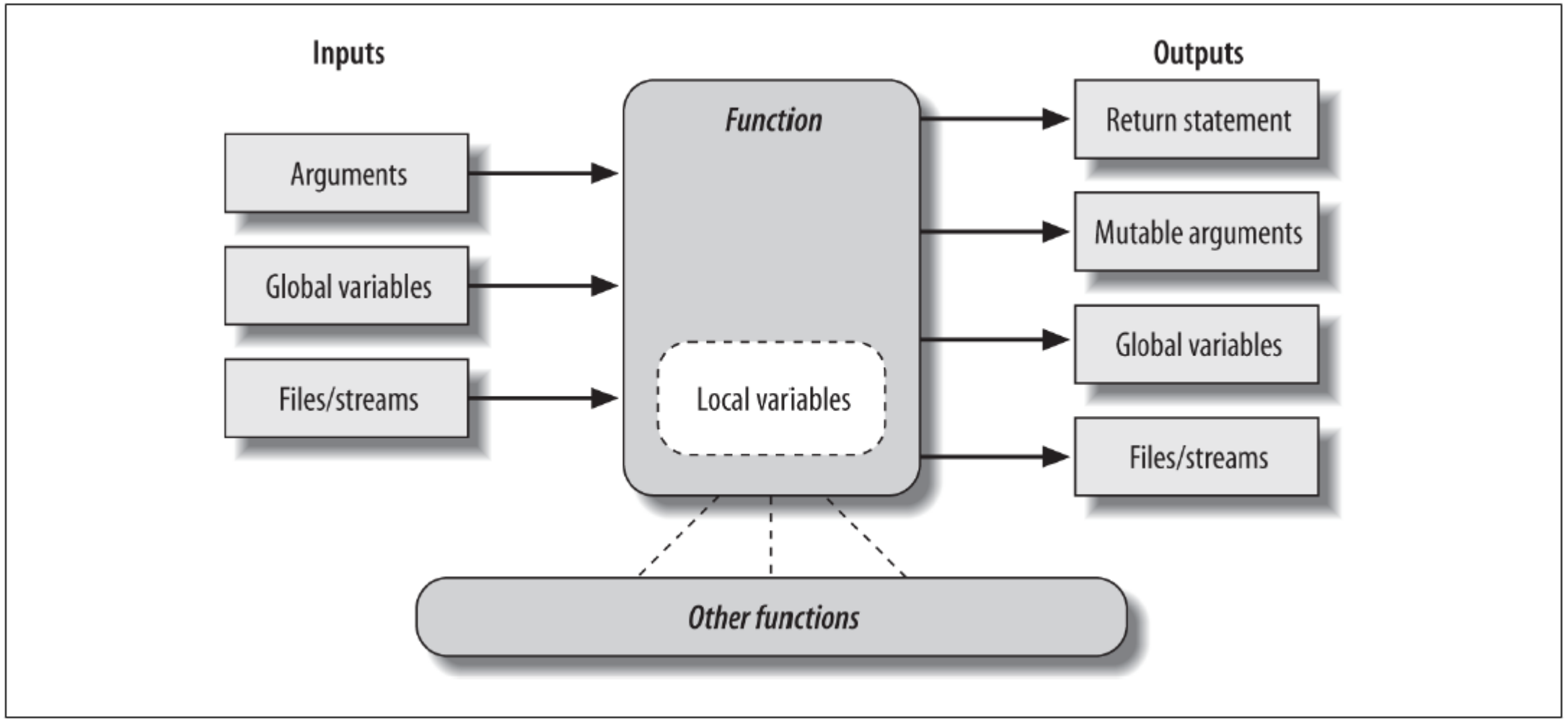
Summary of Functions arguments matching forms:

Syntax	Location	Interpretation
`func(value)`	Caller	Normal argument: matched by position
`func(name=value)`	Caller	Keyword argument: matched by name
`func(*iterable)`	Caller	Pass all objects in *`iterable`* as individual positional arguments
`func(**dict)`	Caller	Pass all key/value pairs in *`dict`* as individual keyword arguments
`def func(name)`	Function	Normal argument: matches any passed value by position or name
`def func(name=value)`	Function	Default argument value, if not passed in the call
`def func(*name)`	Function	Matches and collects remaining positional arguments in a tuple
`def func(**name)`	Function	Matches and collects remaining keyword arguments in a dictionary
`def func(*other, name)`	Function	Arguments that must be passed by keyword only in calls (3.X)
`def func(*, name=value)`	Function	Arguments that must be passed by keyword only in calls (3.X)

Function Design Principles

- Use arguments for input and return for output.
- Use global variables only when truly necessary.
- Do not change mutable arguments unless caller expects it.
- Each function should have a single and unique logic.
- Function size should relatively small.
- Avoid changing variable in another module directly.

Function execution environment

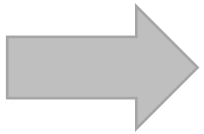


Recursive Function

Definition: A function that calls itself either directly or indirectly in order to loop.

Ex: sum of numbers in a list.

```
def mysum(L):
    if not L:
        return 0
    else:
        return L[0] + mysum(L[1:])

mysum([1, 2, 3, 4, 5])
Ouput: 15
```



```
def mysum(L):
    return 0 if not L else L[0]+mysum(L[1:])

mysum([1, 2, 3, 4, 5])
Ouput: 15
```

```
def normsum(L):
    sum = 0
    for val in L: sum += el
    return sum

normsum([1, 2, 3, 4, 5])
Ouput: 15
```

Function objects, Indirect calls

Every function is also an object.

Ex:

```
def echo(msg):
    print(msg)

echo('Direct Call')
```

Ouput: Direct Call

```
X = echo
X('Indirect Call')
```

Output: Indirect Call

```
L = [(echo, 'spam!'), (echo, 'ham!')]
for func, arg in L:
    func(arg)
```

Ouput:
spam!
ham!

Function's built-in attributes

```
print(echo)

echo.__name__

dir(echo)

echo.__code__

dir(echo.__code__)
```

Function's custom attributes

```
echo.count = 0
echo.count += 1
print(echo.count)

echo.handles = 'Button'
print(echo.handles)

dir(echo) → lists custom attributes too
```

Function Annotations

- Definition: An arbitrary user defined data about function's arguments and result
- It is possible to attach function annotation to function object.
- Python does not do anything with function annotation. They are just display purpose.
- It's completely optional to create and attach annotation.
- If annotation available, can see them using `__annotations__` attribute of function.

```
def enjoy(ch, i, d):
    return ch*i + str(d)

enjoy('Mahesh', 3, 10.5)
Ouput: 'MaheshMaheshMahesh10.5'
```

```
def enjoy(ch: 'name', i: (1, 10), d: float) -> str:
    return ch*i + str(d)

enjoy('Mahesh', 3, 10.5)
Ouput: 'MaheshMaheshMahesh10.5'

enjoy.__annotations__
Output: {'ch': 'name', 'i': (1, 10), 'd': float, 'return': str}

for arg in enjoy.__annotations__:
    print(arg, '=>', enjoy.__annotations__[arg], end='|')
Output: ch => name|i => (1, 10)|d => <class 'float'>|return =>
<class 'str'>|
```

```
#Annotations for default values, return multiple values
def enjoy(ch: 'name' = 'Ganesh', i: (1,
10) = 3, d: float) -> tuple[str, float] |
None :
    return ch*i + str(d), d**i
```

Anonymous Functions: `lambda`

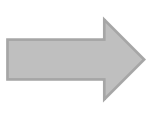
- `lambda` is an **unnamed** function and often used to inline a function definition.
- `lambda` is an expression, not a statement
- `lambda`'s body is a single expression, not a block of statements.
- `lambda` always return a function object
- `lambda` can be nested too.

Syntax:

```
lambda arg1, arg2, …, argN: expression using args
```

Ex:

```
def func(x, y, z):
    return x+y+z

func(2, 3, 4)
Output: 9
```



```
f = lambda x, y, z: x+y+z

f(2, 3, 4)
Ouput: 9
```

Args with default values

```
x = lambda a='fee', b='fie', c='foe': a+b+c

x('wee')
Output: weefiefoe
```

LEGB scope rules applies here too

```
def knights():
    title = 'sir'
    action = (lambda x: title + ' ' + x)
    return action

act = knights()
msg = act('robin')
msg
Output: sir robin
```

```
L = [lambda x: x**2, lambda x: x**3, lambda x: x**4]

for f in L:
    print(f(2))

Output:
4
8
16
```

Generator Functions

- Coded as normal `def` statement that return `generator` object
- but use `yield` statement to return results one at a time.
- Suspend and resume the their state between each result return.

Ex:

Normal Function

```
def buildsquares(n):
    res = []
    for i in range(n):
        res.append(i**2)
    return res

squares = buildsquares(5)
squares
Output: [0, 1, 4, 9, 16]
```

yield vs return

Generator Function

```
def gensquares(N):
    for i in range(N):
        yield i**2

G = gensquares(5)
G
Output: <generator object gensquares at 0x000001B65CB7CDD0>
```

Generator object

Create **iterable** object

```
G = iter(G)
G.__next__(), G.__next__(), G.__next__()

for i in gensquares(5):
    print(i, end=' : ')

Output: 0 : 1 : 4 : 9 : 16 :
```

Manual iterations

Why Generators?

- Generators are better in terms of memory usage and performance in larger programs.
- Very useful when result lists are larger or takes lot of computation to produce each value.

next vs send

When generator sending the result, there is an option to share some information back to generator – using `send()`

Ex:

```
def gen():
    for i in range(10):
        yield i
        print('hi')

G = gen()
next(G)                #it internally calls G.__next__()
Output: 0

next(G)
Output:
hi
1

next(G)
Output:
hi
2
```

```
def gen():
    for i in range(10):
        X = yield i
        print(X)

G = gen()
next(G)                #it internally calls G.__next__()
Output: 0

G.send(77)
Output:
77
1

G.send(88)
Output:
88
3

next(G)
Output:
None
4
```

```
next(G)
Output:
None
2
```

Generator Expression

- The notions of generators and list comprehensions are combined in a new tool: *generator expressions*
- Syntactically just like list comprehensions, but enclosed in ().

Ex:

```
L1 = [i**2 for i in range(4)]
L1
Output: [0, 1, 4, 9]
```

Typical list comprehension

```
G1 = (i**2 for i in range(4))
G1
Output: <generator object <genexpr> at
0x000001E50A9F0740>
```

Typical generator expression

Generator object

```
G = iter(G1)
G.__next__(), G.__next__(), G.__next__()
```

Create **iterable** object

Manual iterations

```
G1 = (i**2 for i in range(4))
list(G1)
Output: [0, 1, 4, 9]
```

Automatic iterations

```
line = 'aa bbb c'
''.join([tx for tx in line.split(' ')])
Output: 'aabbbc'
```

using list comprehension

```
''.join((tx for tx in line.split(' ')))
Output: 'aabbbc'
```

using generator expression

```
''.join(tx for tx in line.split(' '))
Output: 'aabbbc'
```

generator expression parentheses are optional often like this.

Execution Time - `timeit`

- time.time()

```
import time
start = time.time()
for i in range(10000000):
    k = i+1

end = time.time()
clock_time = end - start
print(clock_time)

Output: 1.8710319995880127
```

gets clock time in seconds

- time.process_time()

```
import time
start = time.process_time()
for i in range(10000000):
    k = i+1

end = time.process_time()
exe_time = end - start
print(exe_time)

Output: 1.9375
```

gets CPU execution time in seconds

- timeit module
- datetime module

Execution Time - `timeit`

- time.time()
- time.process_time()
- timeit module
- datetime module

```
import timeit
def addition():
    print('addition:', sum(range(100000)))


result = timeit.timeit(stmt='addition()', globals=globals(), number=5)
print(result)

Output:
addition: 4999950000
addition: 4999950000
addition: 4999950000
addition: 4999950000
addition: 4999950000
0.03184490000001006
```

Namespace which the `stmt` code belongs to.

Avg time for executing `stmt` 5 times

Default runs: 7, default loop count: 10,000

```
import timeit
%timeit [x for x in range(1000)]

Output: 164.9 µs ± 7.15 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each
```

```
import timeit
%timeit -r4 -n15 [x for x in range(1000)]

Output: 97.2 µs ± 3.65 µs per loop (mean ± std. dev. of 4 runs, 15 loops each)
```

Execution Time - `timeit`

- time.time()
- time.process_time()
- timeit module
- datetime module

```
import datetime
start = datetime.datetime.now()
for i in range(10000000):
    k = i+1

end = datetime.datetime.now()
exe_time = end-start
print(exe_time)

Output:
0:00:02.004631
```

gets value in time format : HH:MM:SS

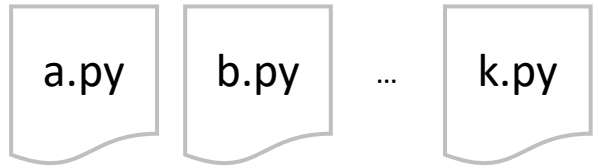
Modules

Module and Uses

Program Hierarchy Recap….

- Programs are composed of modules
- **Modules contains statements**
- **Statements contains expressions**
- **Expressions create and process objects**

Multiple modules in a program

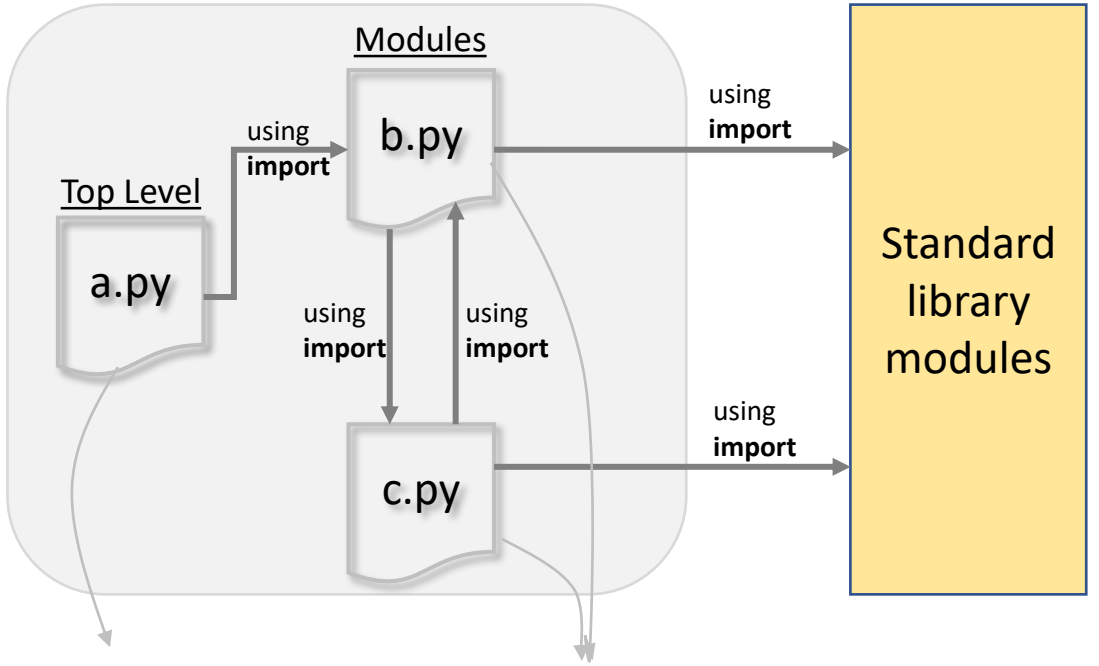


Definition: It's just a **.py** extension file (source file) that contains the some **python statements**

- Code reuse
- Partitioning / grouping the tools/functionalities
- Sharing services or data

Program Structure

- Program contains a *top level* file (a.k.a **script**) and zero or more *supplemental files* (a.k.a **modules**).



b.py

```
import c
def spam(text):
    print(text, 'spam')
```

c.py

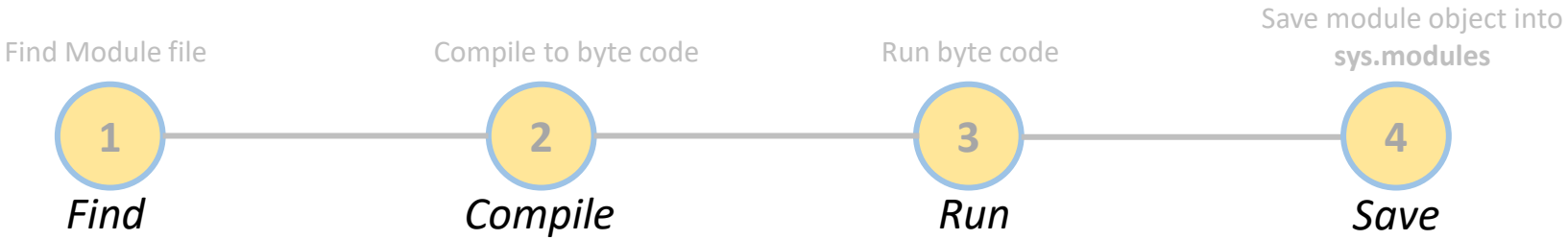
```
import b
def hello(text):
    print('welcome', text)
```

a.py

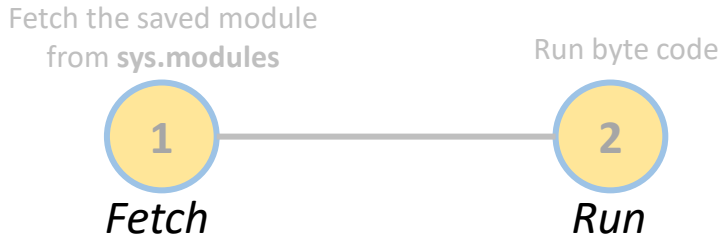
```
import b
b.spam('hi')  # prints "hi spam"
```

How **`import`** works?

First time `import`



Further `import`



How **`import`** works – Step-1 (Find)

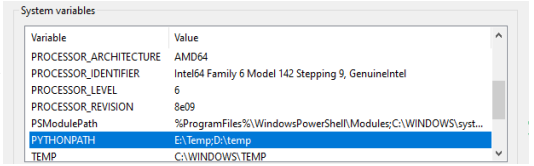
Manual configuration / configuration dependent.

Auto configuration / system dependent.

- Using **Module Search Path**

Home Directory / Current Working Directory.
- Running programs → HD (top level script file path)
- Working in interactive sessions → CWD

PYTHONPATH directories (if set).



Standard library directories. → "standard library" modules path.
C:\Program Files (x86)\Python\Python310\Lib

Contents of any ***.pth*** files (if present). ← a file with .pth extn, contains the list of paths (one path per line)

→ Should be present in either
python installation directory (*C:\Program Files (x86)\Python\Python310*)
or
"site-packages" directory (*C:\Program Files (x86)\Python\Python310\Lib\site-packages*)

site-packages home of third-party extensions. → the "site-packages" sub directory.
C:\Program Files (x86)\Python\Python310\Lib\site-packages

How **import** works – Step-1 (Find)

- All directories in sequence in all 5 components nothing but Module Search Path and saved in **sys.path**
- **sys.path** is a ***mutable list*** of directory name strings.
- Python finds module file in the directory listed in the **sys.path** list from left to right.
- Fetches the first match found and stop search.
- **sys.path** list can be printed:

```
import sys
print(sys.path)
Output:
Ex: ['', 'C:\\Program Files (x86)\\Python\\Python310\\Lib\\idlelib', 'C:\\Program Files (x86)\\Python\\Python310\\python310.zip', 'C:\\Program Files (x86)\\Python\\Python310\\DLLs', 'C:\\Program Files (x86)\\Python\\Python310\\lib', 'C:\\Program Files (x86)\\Python\\Python310', 'C:\\Program Files (x86)\\Python\\Python310\\lib\\site-packages']
```

How **`import`** works – Step-1 (Find)

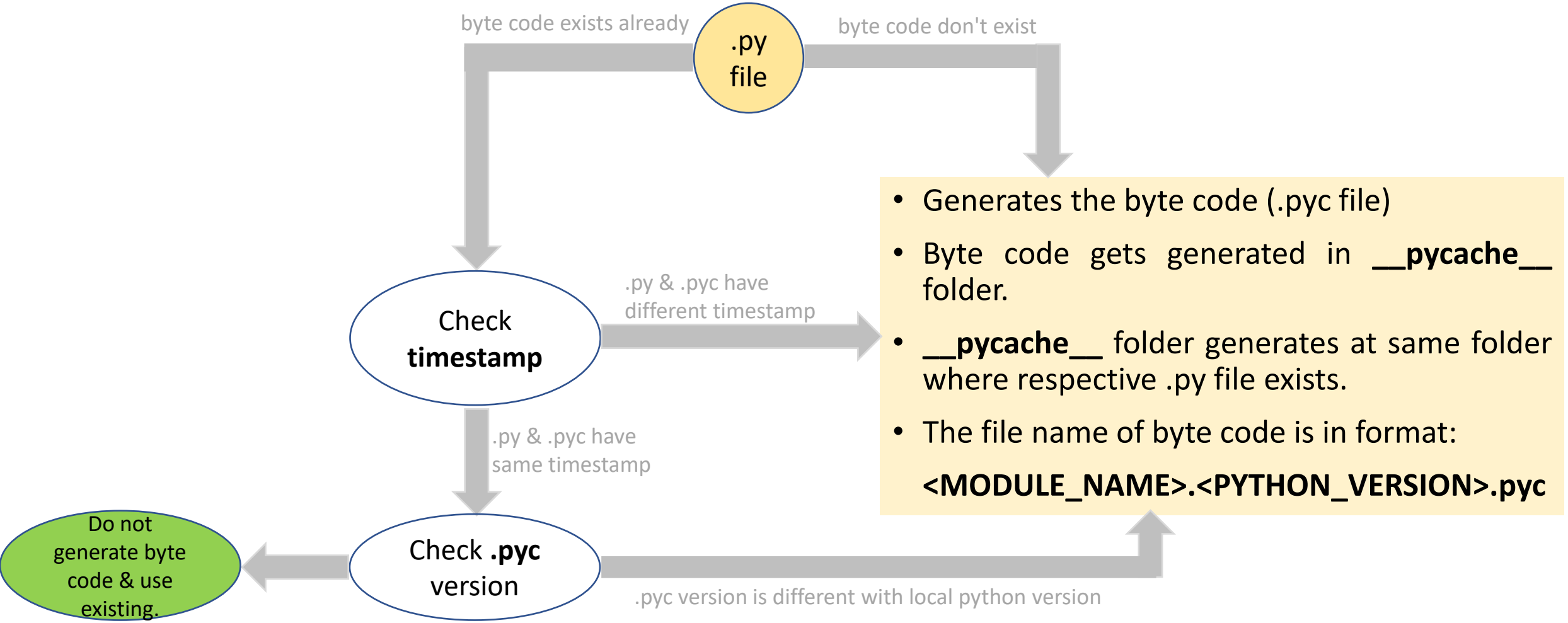
- Python exposes **sys.path** for 2 good reasons:
  - Provides the way to verify the search path settings you made – if you don't see your settings somewhere in this list, you need to recheck your work.
  - Some programs really need to change **sys.path**. Also, `sys.path.append` or `sys.path.insert` often suffice.

    Ex: scripts running on webserver by guest user who has limited access to different folder on server.

- If 2 files (with different allowed extensions, ex: *b.py*, *b.so*) present in the different directories of module search path (sys.path), python picks up the first match found.

- If 2 files (with different allowed extensions, ex: *b.py*, *b.so*) present in the same directory, Python picks up any one randomly.

How **`import`** works – Step-2 (Compile)



How **`import`** works – Step-3 (Run), Step – 4 (Save)

- Simply run the generated byte code (.pyc file)

- Saves the module object in `sys.modules` table.

Modes of `import`

Python always finds the respective module file in **Module Search Path (sys.path)**

The only difference is where Python finds the respective module file (STEP-1 : Find). And the rest of the process (Compile, Run, Save) is the same.

Python always finds the respective module file in **Same Package Folder** of the file that contains this import statement.

Absolute import

Relative import

Fetches the module as a whole. We must qualify to fetch its names.

Copies all names out of the module.

Copies specific names out of the module.

import statement

from statement

from * statement

from . statement

from .. statement

```
import module1
module1.printer('hi')
print(module1.count)
val = module1.getCount()
.
.
.
.
```

```
from module1 import printer
printer('hi')
```

```
from module1 import printer, count
printer('hi')
print(count)
printer(count)
```

```
from module1 import *
print(count, purpose)
printer(getCount())
setCount(112)
.
.
.
.
```

```
from . import module1
print(module1.count)
print(module1.purpose)
printer(module1.getCount())
module1.setCount(112)
.
.
.
.
```

```
from .module1 import count,
purpose
print(count)
print(purpose)
```

```
from .. import module1
print(module1.count)
print(module1.purpose)
printer(module1.getCount())
module1.setCount(112)
.
.
.
.
```

More about `import`

Import happens only once

simple.py

```
print('hello')
spam = 1
```

First Import

```
import simple          # prints: hello
print(simple.spam)     # prints: 1
.
.
.

simple.spam = 2        # change attribute in the loaded module object, available in sys.modules
import simple          # just fetches from already loaded module, available in sys.modules table.
print(simple.spam)     # attribute spam is not reinitialized.
```

Second or Later Import

Reload modules

```
from importlib import reload
reload(simple)         # load module (run module file) and saves in sys.modules table
print(simple.spam)
.
.
.
```

imports can be coded/nested in **if** tests, **def**s, **try** blocks, etc…

```
if x > 10:
    print(x)
    import simple
    value = simple.spam
```

```
def sub(a, b):
    import simple
    if a > b:
        return a-b
    else:
        return simple.spam + a - b
```

More about `import`

Changing mutables of imported modules

small.py

```
x = 1
y = [1, 2]
```

```
from small import x, y
x = 42
y[0] = 42
```

*# copy two names out from **small** module.*
 # changes local x only.
*# changes shared mutable **in place***

```
import small
print(small.x)
print(small.y)
```

*# get module name (**from** doesn't)*
 # prints 1.
 # prints [42, 2]

Cross module name changes

```
from small import x, y
x = 42
```

*# copy two names out from **small** module.*
 # changes local x only, not the x in small module.

```
import small
small.x = 42
```

*# get module name (**from** doesn't)*
*# changes x in loaded module in **sys.modules***

import and **from** equivalence

```
from module import name1, name2
```

Is equivalent to this statement sequence

```
import module
name1 = module.name1
name2 = module.name2
del module
```

Problems with **`from`** import, **`as`** extension

Variables with same names in importing module will be silently overwritten by imported names.

M.py

```
def func():
   print('in M Module')
```

N.py

```
def func():
   print('in N Module')
```

O.py

```
from M import func
from N import func          # This overwrites the one we fetched from M
func()                      # calls N.func only!
```

O.py

```
import M, N                 # Get the whole modules, not their names
M.func()                    # calls func in M module
N.func()                    # Also, calls func from N module. No overwriting.
```

O.py

```
from M import func as mfunc     # Rename uniquely with 'as'
from N import func as nfunc
mfunc(); nfunc()                # calls one or the other
```

from * statement can corrupt namespaces and also difficult to understand imported names.

Even **as** extension does not help here.

MyModule.py

```
from moduleA import *       # copies all names from moduleA to current module
from moduleB import *       # copies all names from moduleB to current module
from moduleC import *       # copies all names from moduleC to current module
func()                      # may overwrite and difficult to understand.
```

O.py

```
import M as m                   # Rename uniquely with 'as'
import N as n                   # Rename uniquely with 'as'
m.func(); n.func()              # calls one or the other
```

Module fundamentals

module1.py

```
count = 100

purpose = 'manage count'

def printer(x):
    print(x)

def getCount():
    return count

def setCount(value):
    global count
    count = value
    def confirm():
        print('count updated!')
```

Module naming also should follow the same rules outlined for normal variables:

- ✓ Syntax: (underscore or letter) + (any number of letters, digits or underscores)
- ✓ Case sensitive.
- ✓ Reserved words are off-limits.

This file is nothing but a **module**.

Only **count**, **printer**, **getCount**, **setCount** are attributes (names associated with this module object (module1)) as they are defined at the top level of this module.

But, **confirm** is not an attribute of this module (module1) as this function is not defined at top level of this module, rather nested in another function.

All names of a module can be accessed using **__dict__** in built attribute of every module object. This always returns a dictionary.

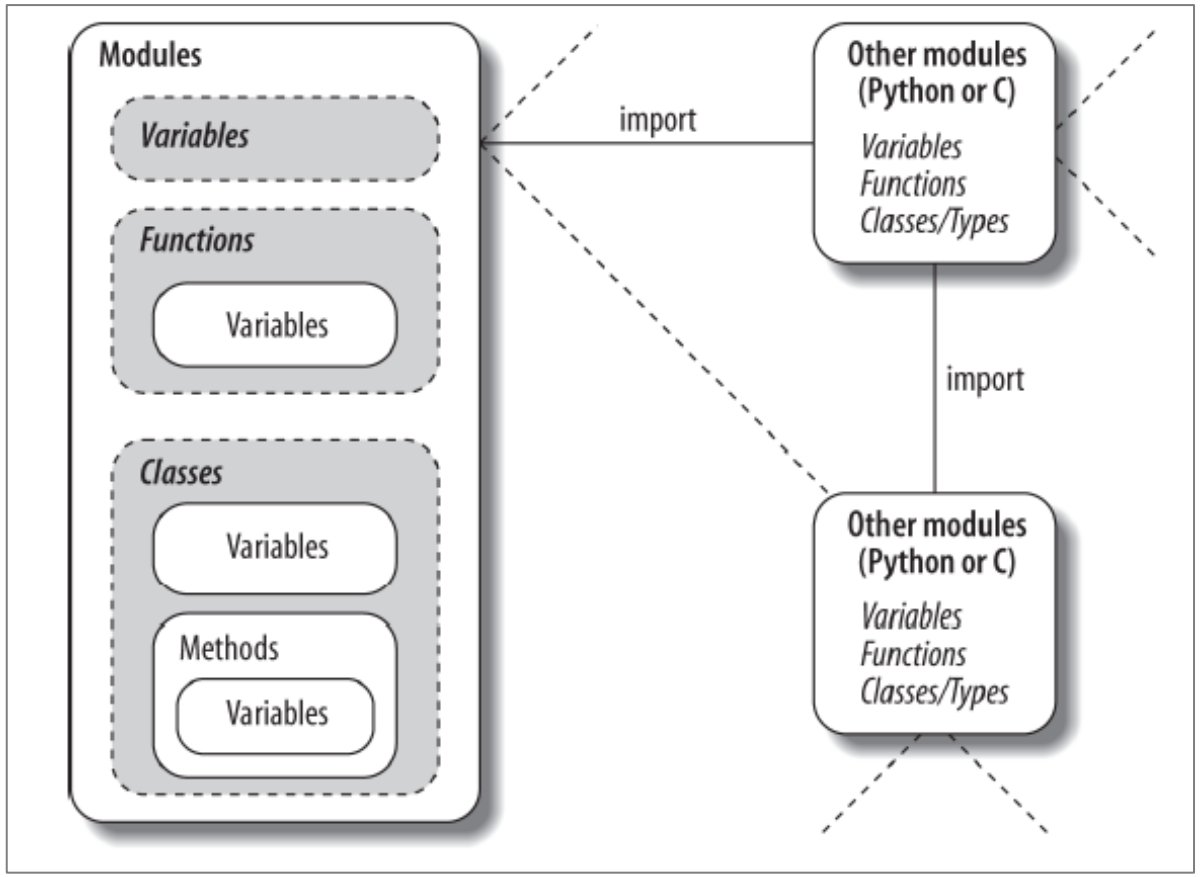
```
print(list(module1.__dict__.keys()))
```

```
O/P: ['__name__', '__doc__', '__package__', '__loader__', '__spec__', '__file__', '__cached__', '__builtins__', 'count', 'purpose', 'printer', 'getCount', 'setCount']
```

Module Design Principles

- Every code in python must be in some module.
- Minimize the module coupling – global variables.
- Modules should rarely change other module's variables.
- Each module should have a specific purpose and have that purpose related code.

Module Execution Environment



Data hiding in modules – `_X, __all__`

`_X` names shall not imported with **`from *`** statement.

another.py
```
from topper import a, _b, c, _d
print(a, _d, c, _b)
Output: 1, 4, 3, 2
```

topper.py
```
a, _b, c, _d = 1, 2, 3, 4
```

someother.py
```
from topper import *
print(a, c)
Output: 1, 3

print(_b)
Output: Name Error: name '_b' is not defined
```

`__all__` is a list contains string names of module. Names present in this list only can be imported with **`from *`** statement.

alls.py
```
a, _b, c, _d = 1, 2, 3, 4
__all__ = ['a', '_d']
```

someother.py
```
from alls import *
print(a, _d)
Output: 1, 4

print(c)
Output: Name Error: name 'c' is not defined
```

another.py
```
import alls
print(alls.a, alls._d, alls.c, alls._b)
Output: 1, 4, 3, 2
```

Module usage modes

- Every module has `__name__` built-in attribute.
- If running module as top level program, `__name__` set to `'__main__'`
- If module is being imported, `__name__` set to module name.

runme.py

```
def tester():
    print('Hi, it a festival today!')

def developer():
    print('Hello, today is holiday.')

if __name__ == '__main__':
    tester()
```

someother.py

```
import runme
runme.tester()
runme.developer()

Output:
Hi, it a festival today!
Hello, today is holiday.
```

```
C:\Users\LENOVO>python runme.py
Hi, it a festival today!
```

Dynamic import

addme.py

```
name = 'Mahesh'

def designer():
    print('Cool, it's my first day.')

def manage():
    print('Hey, join us!.')
```

newmodule.py

```
import addme
addme.designer()
```

```
Output: Cool, it's my first day.
```

newmodule.py

```
from addme import manage
manage()
```

```
Output: Hey, join us!.
```

Using `__import__` built-in function

newmodule.py

```
mod_name = 'addme'
mod_obj = __import__(mod_name)
mod_obj.designer()
```

```
Output: Cool, it's my first day.
```

Using `importlib` module

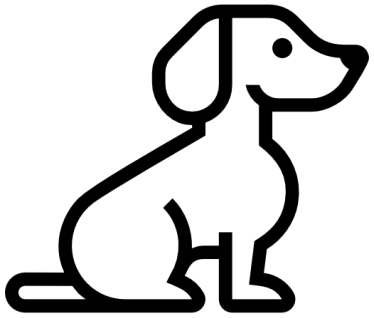
newmodule.py

```
import importlib
mod_name = 'addme'
mod_obj = importlib.import_module(mod_name)
mod_obj.manage()
```

```
Output: Hey, join us!.
```

Classes and OOP

Classes basics, coding classes



Dog

Identified by	Behavior
name	eat()
color	bark()
breed	sleep()
age	play()

mymodule.py

```
count = 0

def designer():
        print('Cool, its my first day.')

def manage():
        print('Hey, join us!.')

class dog:
    name = None
    color = None
    breed = None
    age = 0

    def eat(self):
        print('wrote code logic on how to eat.')

    def bark(self):
        print('wrote code/logic on how to bark.')

    def sleep(self, ihours):
        print('wrote code/logic on how to sleep.')

    def play(self, location):
        print('wrote code/logic on how to play.')
```

A new class 'dog'. This class is also an object of module

Attributes of **dog** class (class attributes)

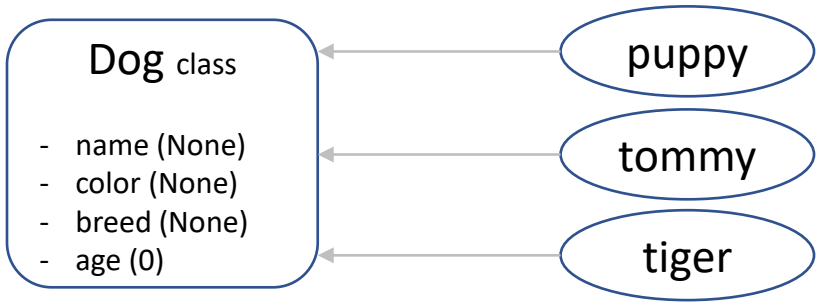
methods of **dog** class

someothermodule.py

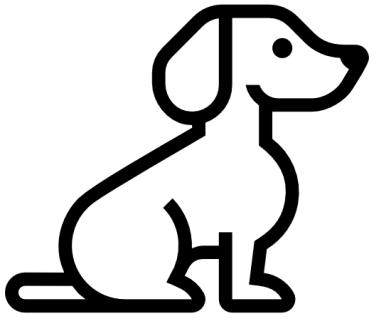
```
from mymodule import dog

puppy = dog()
tommy = dog()
tiger = dog()

print(tiger)
Output: <mymodule.dog object at 0x000001FB90A5AE30>

print(tiger.name, tiger.color, tiger.breed, tiger.age)
Output: None None None 0
```

Every Instance inherits (gets access) class attributes

Accessing **Dog** class attributes using any instance created from Dog class



Classes basics, coding classes



Dog

Identified by	Behavior
name	eat()
color	bark()
breed	sleep()
age	play()

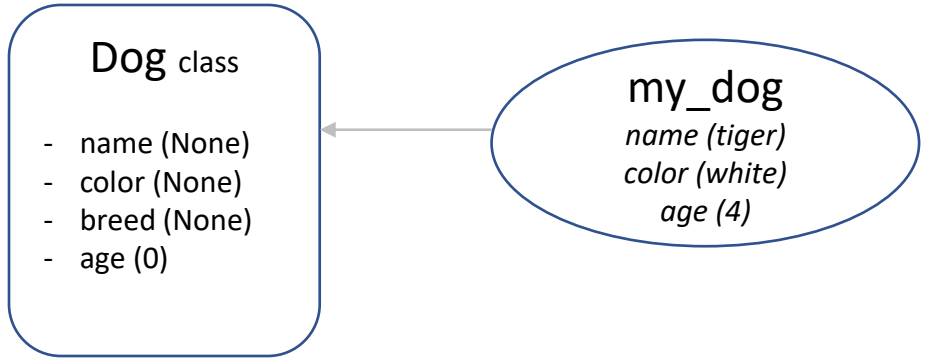
mymodule.py

```
count = 0

def designer():
        print('Cool, its my first day.')

def manage():
        print('Hey, join us!.')

class dog:

    name = None
    color = None
    breed = None
    age = 0

    def set_details(self, dname, dcol, dage):
        self.name, self.color = dname, dcol
        self.age = dage

    def eat(self):
        print('wrote code logic on how to eat.')

    def bark(self):
        print('wrote code/logic on how to bark.')

    def sleep(self, ihours):
        print('wrote code/logic on how to sleep.')

    def play(self, dlocation):
        print('wrote code/logic on how to play.')
```

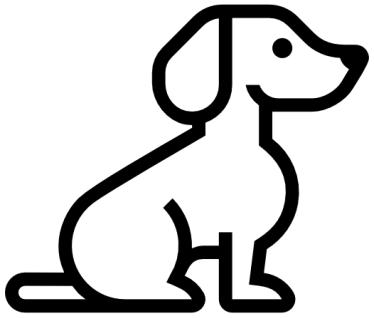
someothermodule.py

```
import mymodule

my_dog = mymodule.dog()
my_dog.set_details('tiger', 'white', 4)


print(my_dog)
Output: <mymodule.dog object at 0x000001FB90A5AE30>

print(my_dog.name, my_dog.color, my_dog.age,
my_dog.breed)
Output: tiger white 4 None
```

With these assignment to attributes of **self**, a new set of these 3 attributes gets created and updated to values in **instance** (instance attributes), not the **class**



Classes basics, coding classes



Dog

Identified by	Behavior
name	eat()
color	bark()
breed	sleep()
age	play()

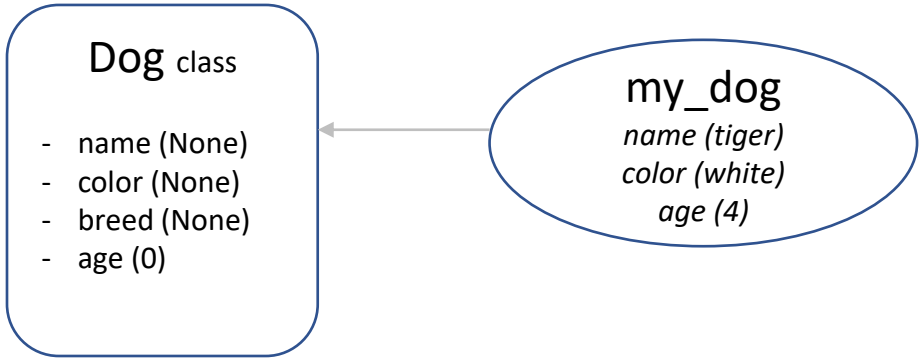
mymodule.py

```
count = 0

def designer():
        print('Cool, its my first day.')

def manage():
        print('Hey, join us!.')

class dog:

    name = None
    color = None
    breed = None
    age = 0

    def __init__(self, dname, dcol, dage):
        self.name, self.color = dname, dcol
        self.age = dage

    def eat(self):
        print('wrote code logic on how to eat.')

    def bark(self):
        print('wrote code/logic on how to bark.')

    def sleep(self, ihours):
        print('wrote code/logic on how to sleep.')

    def play(self, dlocation):
        print('wrote code/logic on how to play.')
```

someothermodule.py

```
from mymodule import dog

my_dog = dog('tiger', 'white', 4)

print(my_dog)
Output: <mymodule.dog object at 0x000001FB90A5AE30>

print(my_dog.name, my_dog.color, my_dog.age,
my_dog.breed)
Output: tiger white 4 None
```

With these assignment to attributes of **self**, a set of these 3 attributes created and updated to values in **instance** (instance attributes), not the **class**



Classes basics, coding classes

World's simplest python class

```
class rec: pass                         #empty class object.

rec.name = 'Bob'                        #attribute belongs to class.
rec.age = 40                            #attribute belongs to class.


x = rec()                               #creating instance of class.
y = rec()                               #and inherits (get access) class attributes

print(x.name, y.name)
Output: Bob Bob

x.name = 'John'                         #creating new name attribute in instance.
print(rec.name, x.name, y.name)
Output: Bob John Bob
```

Modules

- Implement data/logic packages
- Are created with python files (.py files)
- Are used by being imported
- Form the top level in python program structure

Classes

- Implement new full featured objects
- Are created with `class` statements
- Are used by being called
- Always live with in module

Classes basics, coding classes

instance attributes

- Attributes are generated by assignments to `self` attributes in methods.
- Can be accessible by that instance only.
- Can't be accessible by class and other instances of the same class.
- If same attribute present in multiple instances of the class, means, every instance has it's own copy of the attribute and value.

```
class MyAttrs:

    count = 0
    title = 'gives demo on attributes'
    status = 'good'

    def __init__(self, number, info):
        self.status = 'checking'
        self.title = info
        self.display = number
        MyAttrs.count += 1
```

class attributes

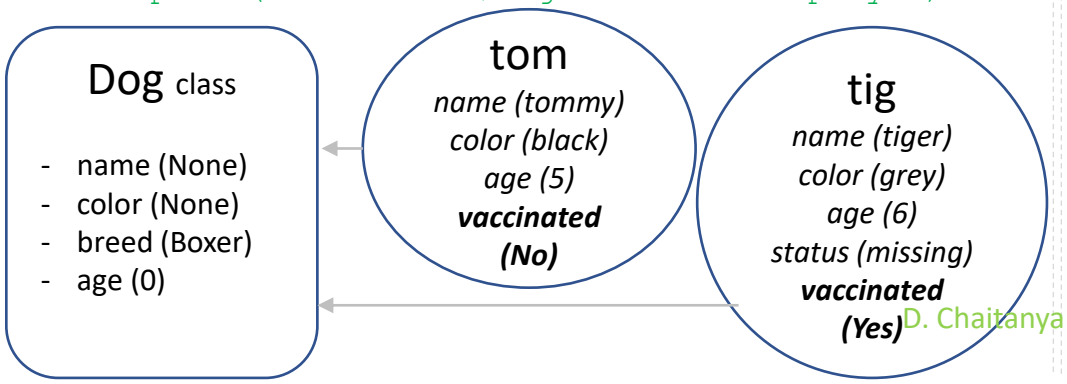
- Attributes created using assignment statements in class.
- Can be accessible by using class or any instance of the class.
- But, can't be updated by using instances of the class.

Summary

- Each class statement generates a new class object
- Each time a class is called, it generates a new instance of the class
- Instances are automatically linked to the classes from which they are created

Classes basics, coding classes

```
class dog:

    name = None
    color = None
    breed = 'Boxer'
    age = 0

    def __init__(self, dname, dcol, dage, dvac):
        self.name, self.color = dname, dcol
        self.age, self.vaccinated = dage, dvac

    def eat(self):
        print('wrote code logic on how to eat.')

    def bark(self):
        print('wrote code/logic on how to bark.')

    def sleep(self, ihours):
        print('wrote code/logic on how to sleep.')

    def play(self, location):
        print('wrote code/logic on how to play.')
```

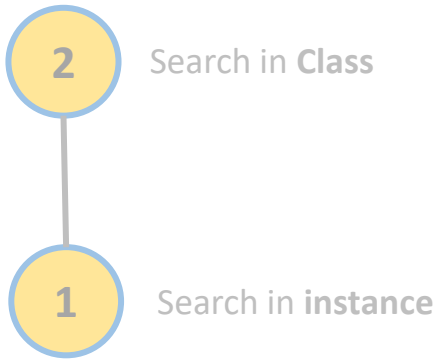



```
tom = dog('tommy', 'black', 5, 'No')
tig = dog('tiger', 'grey', 6, 'Yes')
tig.status = 'missing'

print(tom.name, tom.breed)
Output: tommy Boxer

print(tig.status, tig.breed, tig.vaccinated)
Output: missing Boxer Yes
```

Python performs new and independent search over tree for each attribute fetch expression.

Attribute search logic:

1. Accessing using instance



2. Accessing using class


```
print(tom.name, dog.name)
Output: tommy None

print(tig.name, dog.age, dog.breed)
Output: tiger 0 Boxer
```

Search logic directly search in class as we accessing the attribute using class name.

```
print(dog.vaccinated)
Output: Error!!!
```

Instance attributes can't be accessed by class.

Classes basics, coding classes

```
class dog:

    name = None
    color = None
    breed = 'Boxer'
    age = 0

    def __init__(self, dname, dcol, dage, dvac):
        self.name, self.color = dname, dcol
        self.age, self.vaccinated = dage, dvac

    def eat(self):
        print('wrote code logic on how to eat.')

    def bark(self):
        print('wrote code/logic on how to bark.')

    def sleep(self, ihours):
        print('wrote code/logic on how to sleep.')

    def play(self, location):
        print('wrote code/logic on how to play.')
```

- Logically, Methods provides behavior for instance objects
- Technically, methods work same as functions with a difference that, methods first argument always instance object.

```
tom = dog('tommy', 'black', 5, 'No')

tom.sleep(5)
Output: wrote code/logic on how to sleep.
```

python converts internally to

```
dog.sleep(tom, 5)
```

```
dog.sleep(tom, 5)
Output: wrote code/logic on how to sleep.
```

Python provided Class attributes

```
dir(dog)
['__class__',
 '__dict__',
  .
  .
  .
 'age',
 'bark',
 'play',
 'sleep']
```

dir shows attributes & methods present in class/object also inherited attributes, methods.

```
dog.__name__
Output: 'dog'

dog.__module__
Output: mymodule

dog.__dict__
Output: <check in notebook>
```

Python provided Instance attributes

```
dir(tom)
['__class__',
 '__dict__',
  .
  .
  .
 'age',
 'bark',
 'play',
 'sleep'
 'vaccinated']
```

```
tom.__dict__
Output: {'name':
'tommy', 'color':
'black', 'age': 5,
'vaccinated': 'No'}
```

__dict__ of an instance shows only attributes present in that instance only, does not show attributes present in the instance's class.

```
tom.__class__
__main__.dog

tom.__class__.__name__
Output: 'dog'

tom.__module__
Output: mymodule
```

Classes basics, coding classes

instance methods

- The first argument in method should be always instance (`self`)
- Can be called using instance or class name

```
class methods:

    def imeth(self, x):
        print([self, x])

    @staticmethod
    def smeth(x):
        print([x])

    @classmethod
    def cmeth(cls, x):
        print([cls, x])

    #smeth = staticmethod(smeth)
    #cmeth = classmethod(cmeth)
```

Function decorators

No error only if method is declared as `staticmethod`

static methods

- They are simple functions in class
- No extra argument required to pass.
- By default, can be called using class name only.
- Need to declare it as `static` method if we need to call it using instance too.
- 2 ways to declare method as `static`.

class methods

- The first argument in method should be always class object.
- By default, can be called using class name only. Ex: `methods.cmeth(methods, 15)`
- Need to declare it as `classmethod` if we need to call it using instance as class method.
- 2 ways to declare method as `class`

```
cl_obj = methods()
cl_obj.imeth(5)
Output: [<__main__.methods object at 0x0000021F2D1640D0>, 5]

methods.imeth(cl_obj, 5)
Output: [<__main__.methods object at 0x0000021F2D1640D0>, 5]

methods.smeth(7)
Output: [7]

obj2 = methods()
obj2.smeth(7)
Output: [7]
```

If not defined as `classmethod`, then, this must be the syntax.

```
methods.cmeth(methods, 9)
[<class '__main__.methods'>, 9]

obj3 = methods()
obj3.cmeth(9)  /  methdos.cmeth(9)
Output: [<class '__main__.methods'>, 9]
```

When declared as `classmethod`, can be accessible as class method with instance as well as class (with syntax change)

If not defined as `classmethod`, still no error by accessing it with object. But, python will consider it as instance method while accessing with instance.

Function decorators

Definition : A sort of runtime declaration about the function that follows.

Syntax: starts with @ and followed by a function name (meta function)

```
class C:

    @staticmethod
    def meth(x):
        print([x])
```

Syntax has same effect as

```
class C:

    def meth(x):
        print([x])

    meth = staticmethod(meth)
```

```
class D:

    @classmethod
    def meth(cls, x):
        print([cls.__name__, x])
```

Syntax has same effect as

```
class D:

    def meth(cls, x):
        print([cls.__name__, x])

    meth = classmethod(meth)
```

```
class AClass:

    @property
    def meth(self):
        print(self.__class__.__name__)
```

Results to

Not calling as method, rather calling as attribute.

```
obj = AClass()
obj.meth
Output: AClass
```

User defined function decorators

```
class tracer:

    def __init__(self, func):
        self.calls = 0
        self.func = func

    def __call__(self, *args):
        self.calls += 1
        print(self.calls, self.func.__name__)
        return self.func(*args)

@tracer                          #same as spam = tracer(spam)
def spam(a, b, c):                 means, now spam is a object of tracer.
    return a+b+c


print(spam(1, 2, 3))      #triggers __call__ method
Output: 1 spam

print(spam('a', 'b', 'c'))
Output: 2 spam
```

OOP – Inheritance

- Definition: it is a capability of one class to derive or inherit the properties from another class

- Inheritance allow us to make changes in derived/inherited classes (also called **subclasses**), instead of changing existing classes in place.

- The super classes are listed in parentheses in a `class` header. Ex:
```
class Secondclass(Firstclass):
class Myclass(Firstclass, Seconclass):
```

- Subclass inherits (get access) attributes from their super classes

- Instances inherits (get access) attributes from all accessible classes (class tree hierarchy)

- Each `object.attribute` reference invoke new independent search.

- Logic changes are made by subclassing, not by changing super classes

OOP – Inheritance

Ex1:

```
class FirstClass:
    def setdata(self, value):
        self.data = value

    def display(self):
        print(self.data)


class SecondClass(FirstClass):
    def display(self):
        print('current val =', self.data)


z = SecondClass()
z.setdata(42)
print(z.data, z.display())
Output: 42 current val = 42
```

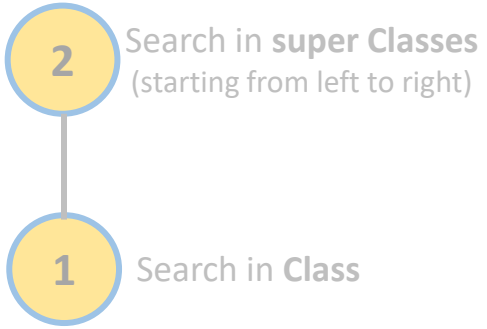
Ex2:

```
class Employee:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay

    def lastName(self):
        return self.name.split(' ')[-1]

    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))

class Manager(Employee):
    def giveRaise(self, percent, bonus=0.1):
        Employee.giveRaise(self, percent+bonus)


bob = Employee('Bob Smith')
sue = Employee('Sue Jones', job='dev', pay=100000)
print(bob.name, bob.job, bob.pay, bob.lastName())
Output: Bob Smith None 0 Smith
print(sue.name, sue.job, sue.pay, sue.lastName())
Output: Sue Jones dev 100000 Jones
sue.giveRaise(0.1)
print(sue.name, sue.job, sue.pay, sue.lastName())
Output: Sue Jones dev 110000 Jones
tom = Manager('Tom Jones', 'mgr', 50000)
print(tom.name, tom.job, tom.pay, tom.lastName())
Output: Tom Jones mgr 50000 Jones
tom.giveRaise(0.1)
print(tom.name, tom.job, tom.pay, tom.lastName())
Output: Tom Jones mgr 60000 Jones
```


Attribute search logic:

1. Accessing using instance

2. Accessing using class



OOP – Inheritance

Ex3:

```
class Employee:
   def __init__(self, name, job=None, pay=0):
      self.name = name
      self.job = job
      self.pay = pay

   def lastName(self):
      return self.name.split(' ')[-1]

   def giveRaise(self, percent):
      self.pay = int(self.pay * (1 + percent))
```

```
class Manager(Employee):
   def __init__(self, name, pay):
      Employee.__init__(self, name, 'mgr', pay)

   def giveRaise(self, percent, bonus=0.1):
      Employee.giveRaise(self, percent+bonus)
```

Customized constructor

```
bob = Employee('Bob Smith')
sue = Employee('Sue Jones', job='dev', pay=100000)
print(bob.name, bob.job, bob.pay, bob.lastName())
Output: Bob Smith Smith

print(sue.name, sue.job, sue.pay, sue.lastName())
Output: Sue Jones dev 100000 Jones

sue.giveRaise(0.1)
print(sue.name, sue.job, sue.pay, sue.lastName())
Output: Sue Jones dev 110000 Jones
```

```
tom = Manager('Tom Jones', 50000)
print(tom.name, tom.job, tom.pay, tom.lastName())
Output: Tom Jones mgr 50000 Jones

tom.giveRaise(0.1)
print(tom.name, tom.job, tom.pay, tom.lastName())
Output: Tom Jones mgr 60000 Jones
```

Deep Insights / Summary

```
class Employee:
   def lastName(self):
      return self.name.split(' ')[-1]
   def giveRaise(self, percent):
      self.pay = int(self.pay * (1 + percent))

Class Manager(Employee):
   def giveRaise(self, percent, bonus=0.1):
      Employee.giveRaise(self, percent+bonus)
   def promote(self, new_job):
      self.job = new_job
      self.giveRaise(0.15, 0.05)
```

 # Inherit
 # Customize
 # Extend

A sub class can…

OOP – Inheritance

Ex4:

```
class A:
   name = 'Mahesh'
   def method(self):
      print('I am in method of Class A')

class B:
   name = 'Suresh'
   age = 40
   def hello(self, msg):
      print(msg)

class C(A, B):
   weight = 56.34
   def sendoff(self):
      print('Bye ', self.name)

c = C()
c.sendoff()
Output: Bye Mahesh
```

Inheriting super classes — Multiple

Ex5:

```
class Super:
   def method(self):
      print('in Super.method')

class Sub(Super):
   def method(self):
      print('starting Sub.method')
      Super.method(self)
      print('ending Sub.method')
```

Can call super class method using it's class name.

```
x = Super()
x.method()
Output: in Super.method

x = Sub()
x.method()
Output:
starting Sub.method
in Super.method
ending Sub.method
```

```
Extender...
starting Extender.method
in Super.method
ending Extender.method


x = Provider()
x.delegate()
Output:
Provider...
in Provider.action
```

Ex6:

```
class Super:
   def method(self):
      print('in Super.method')     # default behavior
   def delegate(self):
      self.action()                # expected to be defined

class Inheritor(Super): pass       # inherit methods from Super

class Replacer(Super):             # Replace method completely
   def method(self):
      print('in Replacer.method')

class Extender(Super):             # Extend method behavior
   def method(self):
      print('starting Extender.method')
      Super.method(self)
      print('ending Extender.method')

class Provider(Super):             # Fill in a required method
   def action(self):
      print('in Provider.action')

for klass in (Inheritor, Replacer, Extender):
   print('\n' + klass.__name__ + '...')
   klass().method()

output:
Inheritor...
in Super.method

Replacer...
in Replacer.method
```

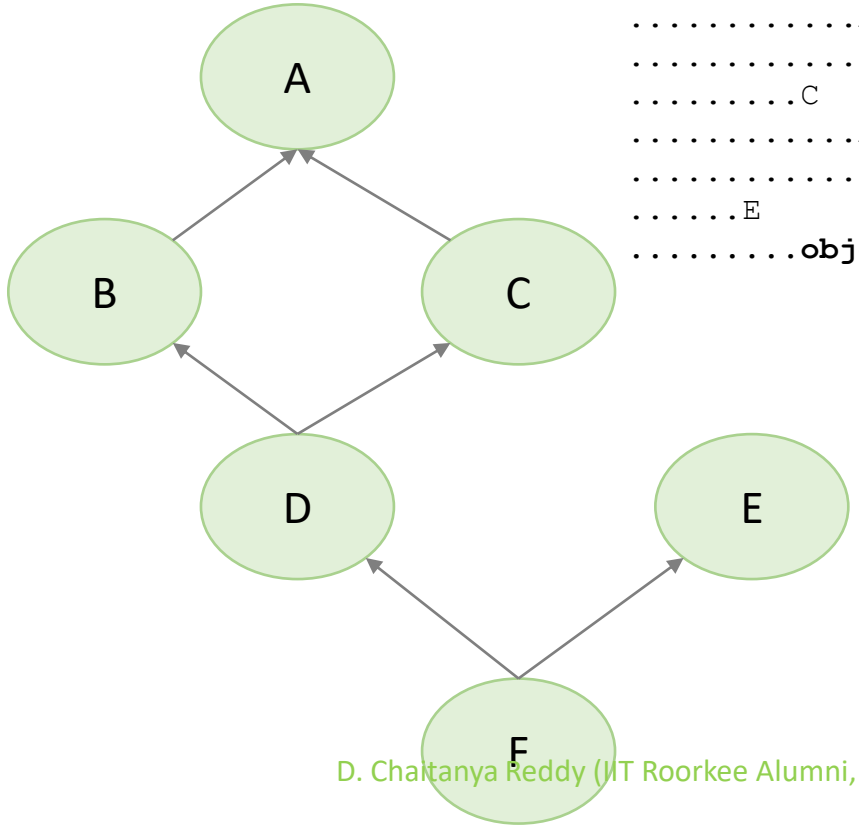
OOP – Inheritance

Ex7:

classtree.py

```
def classtree(cls, indent):
   print('.'*indent + cls.__name__)
   for supercls in cls.__bases__:
      classtree(supercls, indent+3)

def instancetree(inst):
   print('Tree of %s' % inst)
   classtree(inst.__class__, 3)


def selftest():
   class A: pass
   class B(A): pass
   class C(A): pass
   class D(B,C): pass
   class E: pass
   class F(D,E): pass
   instancetree(B())
   instancetree(F())
```

List all classes inherited from




```
selftest()

output:
Tree of <__main__.selftest.<locals>.B object at 0x00000198ECF2E5E0>
...B
......A
.........object
Tree of <__main__.selftest.<locals>.F object at 0x00000198ECF2E5E0>
...F
......D
.........B
............A
..............object
.........C
............A
..............object
......E
.........object
```

object class is super class for all classes if they are not inherited

OOP – Inheritance

Abstract class:

- Definition: Class that expects part of its behavior to be provided by it's subclasses.
- If an expected method is not defined in a subclass, python raises an **undefined name exception** as the search fails.

```
class Super:
   def method(self):
      print('in Super.method')    # default behavior
   def delegate(self):
      self.action()               # expected to be defined

x = Super()
x.delegate()
Output: AttributeError: 'Super' object has no attribute 'action'
```

```
class Super:
   def method(self):
      print('in Super.method')    # default behavior
   def delegate(self):
      self.action()               # expected to be defined

class Sub(Super):
   def printer(self):
      print('hi, I am good.')

x = Super()
x.delegate()
Output: AttributeError: 'Super' object has no attribute 'action'

y = Sub()
y.delegate()
Ouput: AttributeError: 'Sub' object has no attribute 'action'
```

```
class Super:
   def method(self):
      print('in Super.method')    # default behavior
   def delegate(self):
      self.action()               # expected to be defined

class Sub(Super):
   def action(self):
      print('it is fine now.')

y = Sub()
y.delegate()
Ouput: it is fine now.
```

OOP – Composition, Aggregation

Definition: a capability of an object nesting in another object.

```
class Employee:
   def __init__(self, name, job=None, pay=0):
      self.name = name
      self.job = job
      self.pay = pay

   def lastName(self):
      return self.name.split(' ')[-1]

   def giveRaise(self, percent):
      self.pay = int(self.pay * (1 + percent))

class Manager:
   def __init__(self, name, pay):
      self.emp = Employee(name, 'mgr', pay)

   def giveRaise(self, percent, bonus=0.1):
      self.emp.giveRaise(percent+bonus)

x = Manager('Mahesh', 100000)
print(x.emp.name, x.emp.job, x.emp.pay)
Output: Mahesh mgr 100000

x.giveRaise(0.1, 0.15)
print(x.emp.pay)
Output: 125000
```

Embedded a **Employee** object inside **Manager** object.

Definition: a capability of an object has **list** of other objects in it.

```
class Employee:
   def __init__(self, name, job=None, pay=0):
      self.name = name
      self.job = job
      self.pay = pay
   def giveRaise(self, percent):
      self.pay = int(self.pay * (1 + percent))

class Manager(Employee):
   def __init__(self, name, pay):
      Employee.__init__(self, name, 'mgr', pay)
   def giveRaise(self, percent, bonus=0.1):
      Employee.giveRaise(self, percent+bonus)

class Department:
   def __init__(self, *args):
      self.members = list(args)
   def addMember(self, person):
      self.members.append(person)
   def giveRaise(self, percent):
      for person in self.members:
         person.giveRaise(0.1)

bob = Employee('Bob Smith')
sue = Employee('Sue Jones', job='dev', pay=100000)
tom = Manager('Tom Jones', 50000)
development = Department(bob, sue)
development.addMember(tom)
development.giveRaise(0.1)
```

Embedded a list of **Employee** objects inside Department object.

OOP – Polymorphism

Definition: capability that allows us to define methods in the child class with the same name as defined in their parent class.

Which method (from parent or child class) shall be called based on the class from which the object instantiated.

```
class Employee:
   def __init__(self, name, job=None, pay=0):
      self.name = name
      self.job = job
      self.pay = pay

   def lastName(self):
      return self.name.split(' ')[-1]

   def giveRaise(self, percent):
      self.pay = int(self.pay * (1 + percent))

class Manager(Employee):
   def __init__(self, name, pay):
      Employee.__init__(self, name, 'mgr', pay)

   def giveRaise(self, percent, bonus=0.1):
      Employee.giveRaise(self, percent+bonus)
```

```
bob = Employee('Bob Smith')
sue = Employee('Sue Jones', job='dev', pay=100000)
tom = Manager('Tom Jones', 50000)

for obj in (bob, sue, tom):
   obj.giveRaise(0.1)
```

Which class's giveRaise() method shall be called based on the instance's type.

Operator Overloading

```
class Indexer:
   data = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

   @staticmethod
   def get_square(x):
      return x**2


x = Indexer()
print(x.data)
Output: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

print(x.get_square(5))
Output: 25

print(x[4])  -- ????
print(x+4) -- ???
print(x-2) -- ???
print(x) -- ???
print(len(x)) -- ???
print(5 in x) -- ???
print(bool(x)) -- ???
```

- Definition: intercepting (or defining meaning) built-in operations in a class's methods.
- Python automatically invokes these methods when instances of class appear in built-in operations.
- Classes can also overload built-in operations such as printing, function calls, attribute access, etc…
- Overloading makes class instances act more like built-in types
- Overloading is implemented by providing specially named methods in a class.

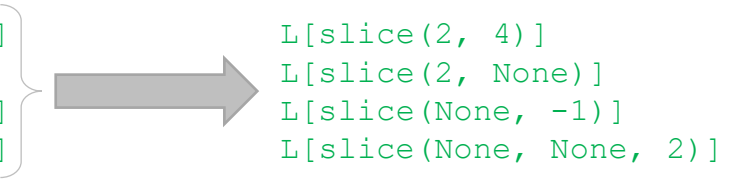
Operator Overloading

```
class Indexer:
   data = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

   def __getitem__(self, index):
      return self.data[index]

x = Indexer()
print(x[4], x[7])
Output: 16 49


y = Indexer()
print(y[2:5], y[6:])
Output: [4, 9, 16] [36, 49, 64, 81, 100]
```

Other scenarios where **__getitem__()** is being called:

```
print(3 in y, 9 in y)          # fallback to __contains__
Output: False True

[n for n in y]                 # fallback to __contains__

(a, b, c, d) = y               # fallback to __iter__ & __next__

list(y), tuple(x)              # fallback to __iter__ & __next__
```

```
L = [5, 6, 7, 8, 9]

L[0]       Output: 5
L[3]       Output: 8
L[2]       Output: 7

L[2:4]     Output: [7, 8]
L[2:]      Output: [7, 8, 9]
```



Operator Overloading


```
class Indexer:
   data = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

   def __getitem__(self, index):
      return self.data[index]

   def __add__(self, value):
      return [x+value for x in self.data]

   def __sub__(self, value):
      return [x-value for x in self.data]
```

Intercepts + operator if and only if the instance is at left side of the operator.

Intercepts – operator if instance is at left side of the operator

```
x = Indexer()
print(x-2)
Output: [-2, -1, 2, 7, 14, 23, 34, 47, 62, 79, 98]

print(x+5)
Output: [5, 6, 9, 14, 21, 30, 41, 54, 69, 86, 105]

print(10+x)
Output: TypeError: unsupported operand type(s) for +:
'int' and 'Indexer'

print(100-a)
Output: unsupported operand type(s) for -: 'int' and
'Indexer'

x += 2; print(x)
Output: [2, 3, 6, 11, 18, 27, 38, 51, 66, 83, 102]

x, y = Indexer(), Indexer()
X += y
Output: TypeError: unsupported operand type(s) for +:
'int' and 'Indexer'
```

Needs **__radd__** method to intercept

Needs **__rsub__** method to intercept

Needs **__iadd__** method to intercept. If not defined __add__ will be used to intercept

```
class Indexer:
   data = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

   def __getitem__(self, index):
      return self.data[index]

   def __add__(self, value):
      Indexer.data = [x+value for x in self.data]
      return self

   def __radd__(self, value):
      Indexer.data = [x+value for x in self.data]
      return self

   def __iadd__(self, value):
      Indexer.data = [x+value for x in self.data if x%2==0]
      print(self.data)
      return self

   def __sub__(self, value):
      Indexer.data = [x-value for x in self.data]
      return self

a = Indexer()
print(10+a)
Output: <__main__.Indexer object at 0x000001439E2EFE50>

print((10+a).data)
Output: [10, 11, 14, 19, 26, 35, 46, 59, 74, 91, 110]

a += 2
print(a.data)
Output: [2, 6, 18, 38, 66, 102]
```

Intercepts + operator if and only if the instance is at right side of the operator and left side operator is not instance of the class.

Intercepts += operator only. If not defined, calls __add__

Also, try this: `a += b`

Operator Overloading


```
class Indexer:
   data = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

x = Indexer()
print(x)
Output: <__main__.Indexer at 0x27718801d60>
```

```
class Indexer:
   data = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

   def __str__(self):
      return 'Indexer with data: %s' % self.data

x = Indexer()
print(x)
Output: Indexer with data: [0, 1, 4, 9, 16, 25, 36,
49, 64, 81, 100]
```

Intercepts bulit-in **print** & **str** functions.

```
class Indexer:
   data = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

   def __str__(self):
      return 'Indexer with data: %s' % self.data

   def __repr__(self):
      return '[Value: %s]' % self.data

>>> import Indexer
>>> x = Indexer()
>>> x
>>> Output: [Value: [0, 1, 4, 9, 16, 25, 36, 49, 64,
81, 100]]
```

Intercepts everything except built-in **print** & **str** functions. Also, works as fallback if NO __**str**__ defined

```
class Indexer:
   data = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

   def __contains__(self, value):
      return value in self.data

   def __bool__(self):
      sum = 0
      for x in self.data:
         sum += x
      return True if sum>200 else False

   def __len__(self): return len(self.data)

   def __call__(self, *pargs, **kargs):
      print('Called:' + pargs, kards)
```

Intercepts **in** operator. If not defined, __getitem__ used as fallback.

Intercepts bool built-in function, any boolean evaluations

Intercepts **len** built-in funciton

Intercepts instance calls

```
k = Indexer()
print(len(k))                    Output: 11
if x: print('hi')                Output: hi
print(5 in x)                    Output: False

m = Indexer()                    # m is a callable object.
m(1, 2, 3)
Output: Called: (1, 2, 3)

m(1, 2, 3, a=4, b=5)
Output: Called: (1, 2, 3) {b=5, a=4}
```

Operator Overloading

Method	Implements	Called for
`__init__`	Constructor	Object creation: `X = Class(args)`
`__del__`	Destructor	Object reclamation of X
`__add__`	Operator +	`X + Y`, `X += Y` if no `__iadd__`
`__or__`	Operator | (bitwise OR)	`X | Y`, `X |= Y` if no `__ior__`
`__repr__`, `__str__`	Printing, conversions	`print(X)`, `repr(X)`, `str(X)`
`__call__`	Function calls	`X(*args, **kargs)`
`__getattr__`	Attribute fetch	`X.undefined`
`__setattr__`	Attribute assignment	`X.any = value`
`__delattr__`	Attribute deletion	`del X.any`
`__getattribute__`	Attribute fetch	`X.any`
`__getitem__`	Indexing, slicing, iteration	`X[key]`, `X[i:j]`, for loops and other iterations if no `__iter__`
`__setitem__`	Index and slice assignment	`X[key] = value`, `X[i:j] = iterable`
`__delitem__`	Index and slice deletion	`del X[key]`, `del X[i:j]`
`__len__`	Length	`len(X)`, truth tests if no `__bool__`
`__bool__`	Boolean tests	`bool(X)`, truth tests (named `__nonzero__` in 2.X)
`__lt__`, `__gt__`, `__le__`, `__ge__`, `__eq__`, `__ne__`	Comparisons	`X < Y`, `X > Y`, `X <= Y`, `X >= Y`, `X == Y`, `X != Y` (or else `__cmp__` in 2.X only)

Operator Overloading

Method	Implements	Called for
`__radd__`	Right-side operators	`Other + X`
`__iadd__`	In-place augmented operators	`X += Y` (or else `__add__`)
`__iter__`, `__next__`	Iteration contexts	`I=iter(X)`, `next(I)`; `for` loops, `in` if no `__contains__`, all comprehensions, `map(F,X)`, others (`__next__` is named `next` in 2.X)
`__contains__`	Membership test	`item in X` (any iterable)
`__index__`	Integer value	`hex(X)`, `bin(X)`, `oct(X)`, `O[X]`, `O[X:]` (replaces 2.X `__oct__`, `__hex__`)
`__enter__`, `__exit__`	Context manager	`with obj as var:`
`__get__`, `__set__`, `__delete__`	Descriptor attributes	`X.attr`, `X.attr = value`, `del X.attr`
`__new__`	Creation	Object creation, before `__init__`

Classes - Special Points

- Every instance is made from a class and **`type(obj)`** *built-in* function tells us **`obj`** instance is made from which class.

- We can also use **`obj.__class__`** to know the class from which the **`obj`** instance is made.

- Every class is also an object (instance). All classes are instantiated from class **`type`** (instances of **`type`** class).

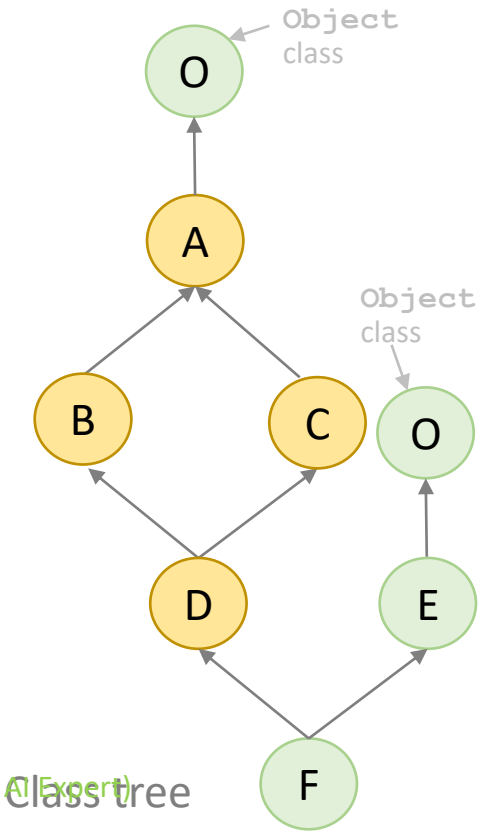
```
class D: pass

I = D()
print(type(I), I.__class__)
Output: <class '__main__.D'> <class '__main__.D'>
```

```
print(type(D), D.__class__)
Output: <class 'type'> <class 'type'>
```

- If no super class is mentioned, every class by default, inherits from **`object`** super class though the **`object`** class is NOT mentioned as super class explicitly.

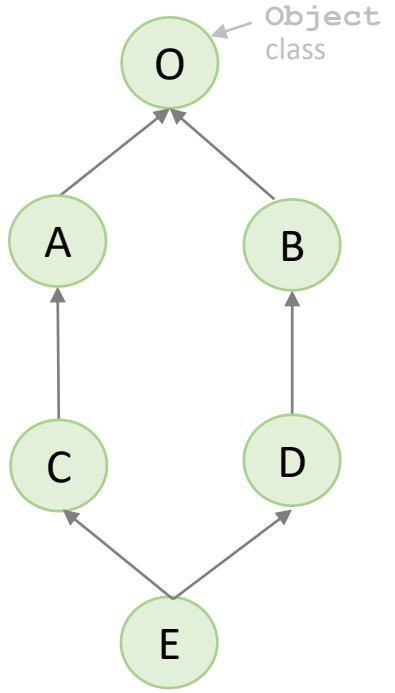
- Attribute/method search logic in inheritance search shall vary based on the **diamond**, **non-diamond** class trees formed in tree hierarchy.

```
class A: pass
class B(A): pass
class C(A): pass
class D(B,C): pass
class E: pass
class F(D,E): pass
```



Class tree

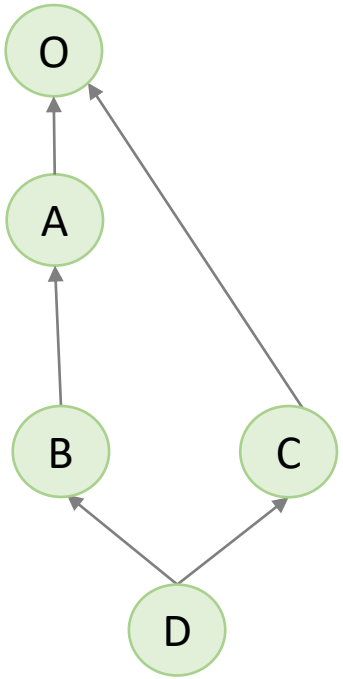
```
class A: pass
class B: pass
class C(A): pass
class D(B): pass
class E(C,D): pass
```



Class tree

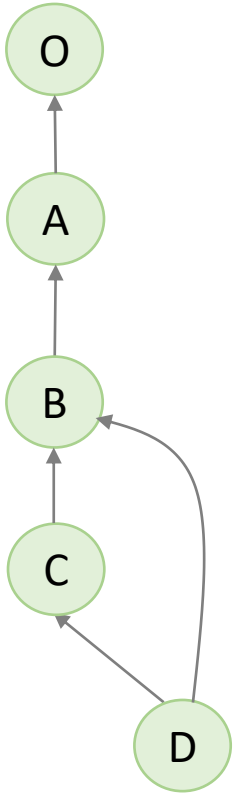
Classes - Special Points

```
class A: pass
class B(A): pass
class C: pass
class D(B,C): pass
```



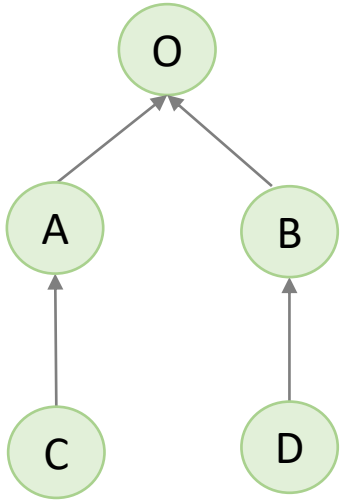
Class tree

```
class A: pass
class B(A): pass
class C(B): pass
class D(B,C): pass
```



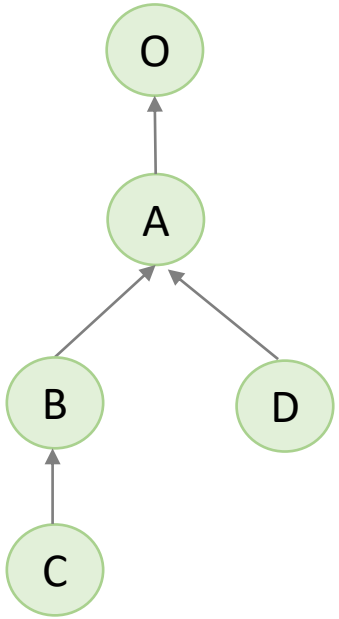
Class tree

```
class A: pass
class B: pass
class C(A): pass
class D(B: pass
```



Class tree

```
class A: pass
class B(A): pass
class C(B): pass
class D(A): pass
```



Class tree

Inheritance search logic

Recap….

Search logic name: MRO (Method Resolution Order)

Attribute search logic:

1. Accessing using instance

3. Search in super Classes (**starting from left to right**)

2. Search in **Class**

1. Search in **instance**

This inheritance search logic is applicable only for **Diamond shape** class trees

2. Accessing using class

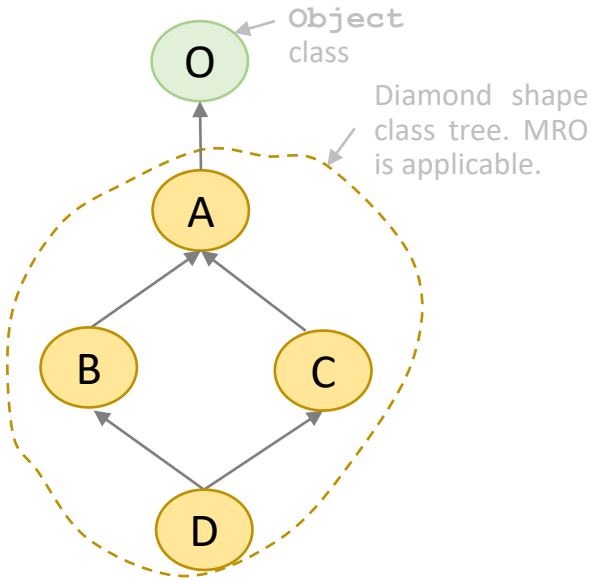
2. Search in super Classes (**starting from left to right**)

1. Search in **Class**

Ex:

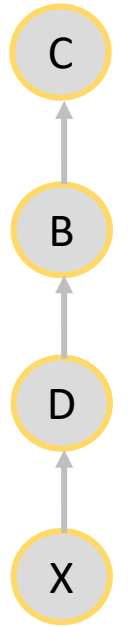
```
class A: attr=1
class B(A): pass
class C(A): attr=3
class D(B,C): pass

X = D()
print(X.attr)
```

Output: 3



Class tree

Search Order



Inheritance search logic

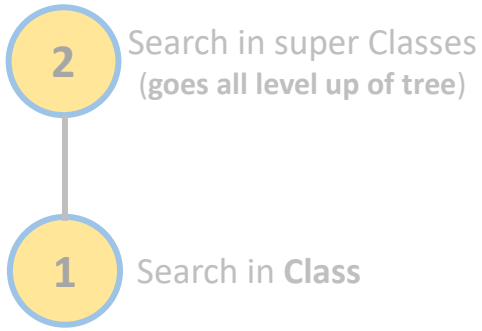
Attribute search logic:

Search logic name: DFLR (Depth First Left Right)

1. Accessing using instance

This inheritance search logic is applicable only for **Non Diamond shape** class trees

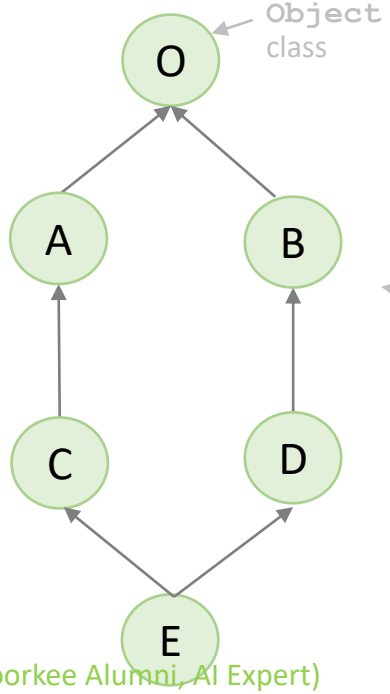
2. Accessing using class



Ex:

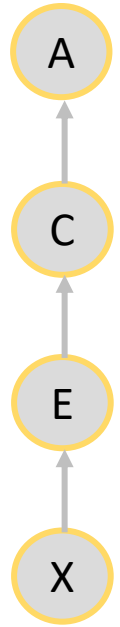
```
class A: attr=1
class B: attr=2
class C(A): pass
class D(B): attr=4
class E(C,D): pass
```

```
X = E()
X.attr
Output: 1
```



Non Diamond shape class tree. DFLR is applicable.

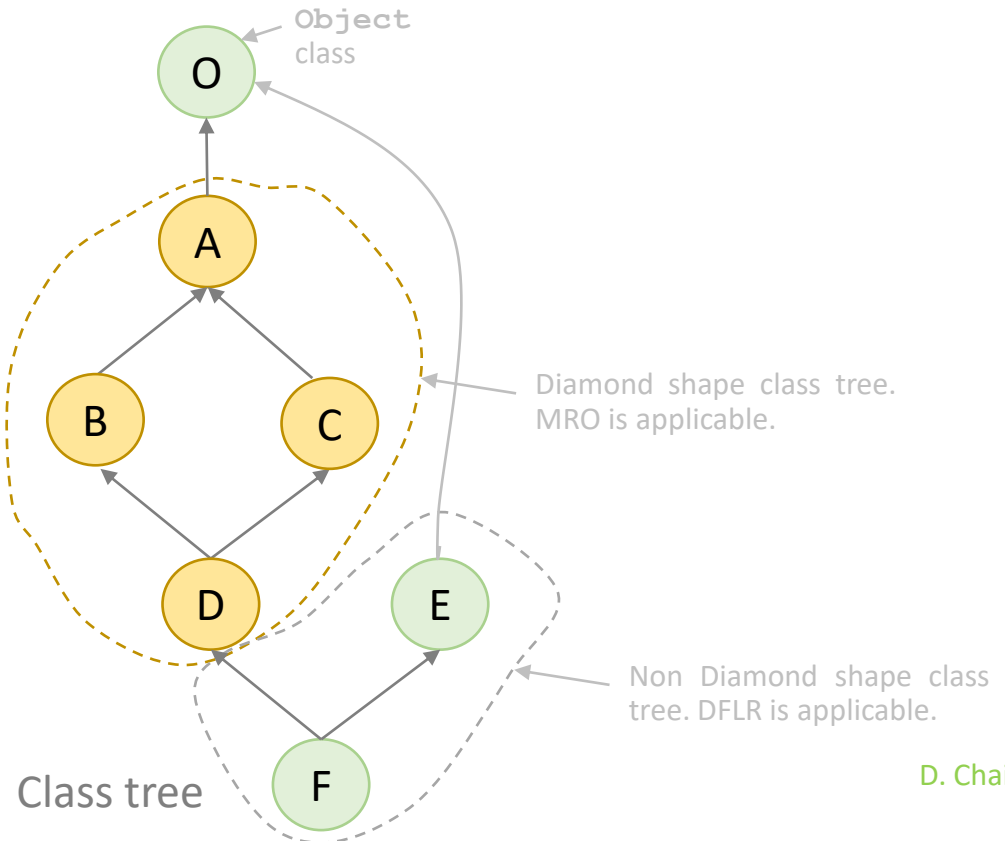
Search Order



Inheritance search logic

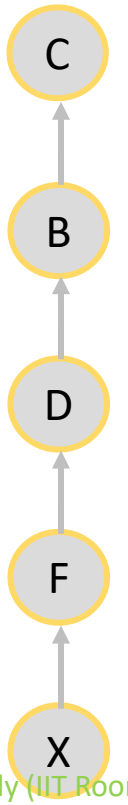
Ex:

```
class A: attr=1
class B(A): pass
class C(A): attr=3
class D(B,C): pass
class E: attr=5
class F(D,E): pass
```



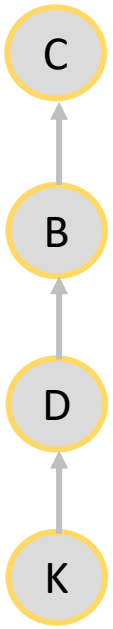

```
X = F()
X.attr
Output: 3
```

Search Order




```
K = D()
K.attr
Output: 3
```

Search Order




```
print(F.__mro__)
Output:
(__main__.F,
 __main__.D,
 __main__.B,
 __main__.C,
 __main__.A,
 __main__.E,
 object)
```

```
print(D.__mro__)
Output:
(__main__.D,
 __main__.B,
 __main__.C,
 __main__.A,
 object)
```

Advanced Classes: Pseudo private class attribute

```
class C1:
    def meth1(self): self.X = 88
    def meth2(self): print(self.X)

class C2:
    def metha(self): self.X = 99
    def methb(self): print(self.X)

class C3(C1, C2):
    def __init__(self): self.X = 111


I = C3()
print(I.X)
Output: 111

I.meth1()
print(I.X)
Output: 88

I.metha()
print(I.X)
Output: 99
```

- **Name Mangling:** Any attribute or method starts with "__" and not end with "__" are automatically expanded to include the name of enclosing class at their front.

```
class spam:
    __X = 88

I = spam()
print(I.__X)
Output: AttributeError: 'spam' object has no attribute '__X'

print(I._spam__X)
Output: 88
```

- Name mangling happens only for names (attributes/methods) appear inside a class statement's code.
- Name mangling is applicable for both class attributes and method and instance attributes which are prefixed with "__"
- These mangled names sometimes **misleadingly** called as **private** attributes. But, they are really not private attributes. Hence, named as **Pseudo private** attributes.
- There is no restriction for other classes to access these attributes and hence these are not really private attributes.

- The purpose of these mangled attributes is to avoid attribute name collisions

Advanced Classes: Pseudo private class attribute

```
class C1:
    def set_value(self, val): self.__X = val
    def get_value(self): return self.__X

class C2:
    def set_info(self, info): self.__X = info
    def get_info(self): return self.__X

class C3(C1, C2):
    def __init__(self): self.__X = 111


I = C3()
I._C1__X = 25
I._C2__X = 42


print(I._C1__X, I._C2__X, I._C3__X)
Output: 25, 42, 111


J = C3()
J.set_value(55)
J.set_info(10)


print(J.get_value(), J.get_info())
Output: 55, 10
```

Mangled names are NOT required use to access the attributes inside the class's statement.

Mangled names are required to access the attributes outside the class's statement.

One way of accessing the pseudo private attributes from outside of the class.

Other way of accessing the pseudo private attributes from outside of the class.

Encapsulation

Advanced Classes: Attribute namespaces, Slots

Attribute Namespaces:

__dict__
Attribute name space

```
class limiter:
    min_age = 34
    count_limit = 100
    max_age = 72

    def __init__(self, username, designation):
        self.name = username
        self.job = designation
```

Class attributes

Instance attributes

```
print(limiter.__dict__)
Output:
{'__module__': '__main__', 'min_age': 34, 'count_limit': 100,
'max_age': 72, '__init__': <function limiter.__init__ at
0x0000019AD7DDA040>, '__dict__': <attribute '__dict__' of
'limiter' objects>, '__weakref__': <attribute '__weakref__' of
'limiter' objects>, '__doc__': None}
```

Shows list of all **class** attributes (both user defined and python default provided

```
x = limiter('mahesh', 'developer')
print(x.__dict__)
Output: {'name': 'mahesh', 'job': 'developer'}
```

Shows list of all **x instance** attributes

```
y = limiter('david', 'manager')
print(y.__dict__)
Output: {'name': 'david', 'job': 'manager'}
```

Shows list of all **y instance** attributes

Slots: Technique that allows us to assign a sequence/list of attribute names (in string format) to a special **__slots__** class attribute in order to limit and fix the attributes present in every instance of class.

Ex:

Every instance of **limiter** class must have only 3 attribute with these names in it with no default values assigned.

But, **limiter** class allows attributes (class attributes) with any names.

```
class limiter:
    __slots__ = ['age', 'name', 'job']

    def __init__(self, username, designation):
        self.name = username
        self.job = designation


x = limiter('mahesh', 'developer')
print(x.name, x.job)
Output: mahesh developer

print(x.age)
AttributeError : age
```

Instance attribute must also assigned with some value before being used/referenced.

```
x.age = 40
print(x.age)
Output: 40
```

Another Instance attribute.

```
x.city = 'Mumbai'
Attribute Error: 'limiter' object has no attribute 'city'.
```

Instance can't have any attribute apart from defined slot attributes: age, name, job.

Advanced Classes: Attribute namespaces, Slots

```
class slotful:
    __slots__ = ['age', 'name', 'job']
    lmt = 120
    ret_age = 75

    def __init__(self, username, designation):
        self.name = username
        self.job = designation

print(slotful.__dict__)
Output:
{'__module__': '__main__', '__slots__': ['age', 'name', 'job'],
'lmt': 120, 'ret_age': 75, '__init__': <function
slotful.__init__ at 0x0000019AD98B28B0>, 'age': <member 'age'
of 'slotful' objects>, 'job': <member 'job' of 'slotful'
objects>, 'name': <member 'name' of 'slotful' objects>,
'__doc__': None}

x = slotful('suresh', 'tester')
print(x.__dict__)
Output: AttributeError: 'slotful' object has no attribute
'__dict__'

print(slotful.__slots__)
Output: ['age', 'name', 'job']

print(x.__slots__)
Output: ['age', 'name', 'job']
```

Class that contains __slots__ can not have attribute namespace (__dict__) by default.

```
class slotdic:
    __slots__ = ['age', 'name', 'job', '__dict__']
    lmt = 120
    ret_age = 75

    def __init__(self, username, designation):
        self.name = username
        self.job = designation
        self.country = 'india'

print(slotdic.__dict__)
Output:
{'__module__': '__main__', '__slots__': ['age', 'name', 'job',
'__dict__'], 'lmt': 120, 'ret_age': 75, '__init__': <function
slotdic.__init__ at 0x0000019AD98FE5E0>, 'age': <member 'age'
of 'slotdic' objects>, 'job': <member 'job' of 'slotdic'
objects>, 'name': <member 'name' of 'slotdic' objects>,
'__dict__': <attribute '__dict__' of 'slotdic' objects>,
'__doc__': None}

z = slotdic('naresh', 'team leader')
print(z.__dict__)
Output: {'country': 'india'}

print(slotdic.__slots__)
Output: ['age', 'name', 'job', '__dict__']

print(z.__slots__)
Output: ['age', 'name', 'job', '__dict__']
```

Advanced Classes: Attribute namespaces, Slots

```
class E:
   __slots__ = ['age', 'name']

class D(E):
   __slots__ = ['job', '__dict__']


X = D()
X.age = 45; X.name='Ramesh'; X.job = 'anchor'

print(E.__dict__)
Output: {'__module__': '__main__', '__slots__': ['age',
'name'], 'age': <member 'age' of 'E' objects>, 'name':
<member 'name' of 'E' objects>, '__doc__': None}

print(D.__dict__)
Output: {'__module__': '__main__', '__slots__': ['job',
'__dict__'], 'job': <member 'job' of 'D' objects>,
'__dict__': <attribute '__dict__' of 'D' objects>,
'__doc__': None}

print(X.__dict__)
Output: {'age': 45, 'name': 'Ramesh', 'job': 'anchor'}


print(E.__slots__)
Output: ['age', 'name']

print(D.__slots__) or print(X.__slots__)
Output: ['job', '__dict__']
```

Slots Usage Rules

- Slots in sub classes are meaningless when absent in its super classes.

```
class C: pass
class D(C): __slots__ = ['a']

X = D()
X.a = 1; X.b = 2
print(X.__dict__)
Output: {'b': 2}
```

- Slots in super classes are meaningless when absent in its sub classes

```
class C: __slots__ = ['a']
class D(C): pass

X = D()
X.a = 1; X.b = 2
print(X.__dict__)
Output: {'b': 2}
```

Advanced Classes: Attribute namespaces, Slots

Slots Usage Rules

- Overriding Slots in sub classes makes super classes slots meaningless.

```
class C: __slots__ = ['a']
class D(C): __slots__ = ['a']

X = D()
X.a = 1
print(X.__slots__)
Output: ['a']
```

- Slots prevents/conflicts class attributes.

```
class C:
    __slots__ = ['a']
    a = 99
```

Error: **'a'** in __slots__ conflicts with class variable

Benefits of `Slots`

- Python reserves just enough space in each instance to hold a value for each slot attribute along with inherited attributes in the super class.
- Helps python to mange the memory very effectively.
- Speeds up the code execution.
- Best suited for scenarios where large number of instances with only few attributes to be created.

Advanced Classes : super()

Calling super class's method

```
class C:
    def act(self):
        print('spam')

class D(C):
    def act(self):
        C.act(self)
        print('eggs')

X = D()
X.act()
Output:
spam
eggs
```

Syntax to call super class's method using **super class name**.

Need to pass `self` as calling super method with class name.

```
class C:
    def act(self):
        print('spam')

class D(C):
    def act(self):
        super().act()
        print('eggs')

X = D()
X.act()
Output:
spam
eggs
```

Syntax to call super class's method using **super()**

No need to pass `self`, because method is being called with magical proxy super class object.

With multiple super classes

```
class A:
    def act(self): print('A')
class B:
    def act(self): print('B')
class C(A, B):
    def act(self):
        super().act()
        print('C')

X = C()
X.act()
Output:
A
C
```

Which super class's act() method will be called?

Rule: **Python** always picks the left most super class having that method. Technically, the 1st class in MRO search having that method.

```
class A:
    def act(self): print('A')
class B:
    def act(self): print('B')
class C(A, B):
    def act(self):
        B.act(self)
        print('C')

X = C()
X.act()
Output:
B
C
```

Best Practice:

1) If the class have only single super class and no multiple super classes comes in future too, user super() to call super class methods.

2) If the class have multiple super classes, always use class names to call specific super class method.

Exceptions

Exceptions

- If any error found while running code, python triggers that error as exception.
- The triggered exception can be intercepted/caught by code.

```
def fetcher(obj, index):
    return obj[index]

x = 'spam'
fetcher(x, 3)
Output: 'm'
```

Default Exception Handler

```
fetcher(x, 4)
Output:
IndexError
Traceback (most recent call last)
Input In [2], in <cell line: 2>()
      1 x = 'spam'
----> 2 fetcher(x, 4)

Input In [1], in fetcher(obj, index)
      1 def fetcher(obj, index):
----> 2     return obj[index]

IndexError: string index out of range
```

```
def fetcher(obj, index):
    return obj[index]

x = 'spam'
try:
    fetcher(x, 4)
except IndexError:
    print('got Index err')

Output: got Index err
```

Own exception handler

Exceptions

try statement syntax

```
try:
    statement(s)                # run this main action first
except name1:                   # catch a specific exception only.
    statement(s)                # run if name1 is raised during try block
except (name2, name3):          # catch any of the listed exceptions
    statement(s)                # run if any of these exceptions occurs
except name4 as var:            # catch a specific exception and assign it's instance
    statement(s)                # run if name4 is raised, assign instance raised to var
except (name5, name6) as var:   # catch any of the listed exceptions and assign it's instance
    statement(s)                # run if any of these exception occurs, assign instance raised to var
except:                         # catch all (or all other) exceptions
    statement(s)                # run for all other exceptions raised
else:
    statement(s)                # run if no exception was raised during try block
finally:
    statement(s)                # always run this code in any case.
```

try block which contains the code which may causes some errors while running it.

A **try** block can associate with either with zero or one or multiple exception handlers (**except** block).

Can't exists without associated **try**

if a **single except** block is present, python checks whether the `except` block handles the raised exception or not.

If **multiple except** blocks are present, python finds the first matching `except` block to handle the exception and the further except blocks are ignored.

default **except:** must be always at last in except block.

Can have zero or one **else** block

Can't exists without at least one exception handler

Can have optional **else** block if at least one exception handler exists.

Code in **else** block shall execute if and only if no exception raised in **try** block code.

Can have zero or one **finally** block

Must exists if **try** does not have at least one exception handler.

Can be optional if **try** have at least one exception handler.

Exceptions

Ex:

```
def action(x, y):
    print(x + y)


try:
    action([1, 2, 3], 6)
except NameError:
    print('got Name Error')
except TypeError as err:
    print('got Type Error.')
except:
    print('got some different error')
else:
    print('No Error found.')
finally:
    print('taking care of clean up code here')
print('After the try statement.')
```

```
Output:
got Type Error.
taking care of clean up code here
After the try statement.
```

Exceptions – `try` statement formats

Possible-1

```
try:
    statement(s)
except:
    statement(s)
```

Possible-2

```
try:
    statement(s)
except name1:
    statement(s)
except (name2, name3):
    statement(s)
except name4 as var:
    statement(s)
except (name5, name6) as var:
    statement(s)
except:
    statement(s)
```

Possible-3

```
try:
    statement(s)
except name1:
    statement(s)
else:
    statement(s)
```

Possible-4

```
try:
    statement(s)
except (name2, name3):
    statement(s)
except name4 as var:
    statement(s)
except name1:
    statement(s)
else:
    statement(s)
```

Possible-5

```
try:
    statement(s)
except name1:
    statement(s)
else:
    statement(s)
finally:
    statement(s)
```

Possible-6

```
try:
    statement(s)
finally:
    statement(s)
```

Possible-7

```
try:
    statement(s)
except (name5, name6) as var:
    statement(s)
finally:
    statement(s)
```

Possible-8

```
try:
    statement(s)
except (name5, name6) as var:
    statement(s)
except Exception:
    statement(s)
except:
    statements(s)
finally:
    statement(s)
```

Exceptions – `try` statement formats

Possible-9

```
try:
    statement(s)
except (name5, name6) as var:
    statement(s)
except name1:
    statement(s)
except (name2, name3):
    statement(s)
except name4 as var:
    statement(s)
else:
    statement(s)
finally:
    statement(s)
```

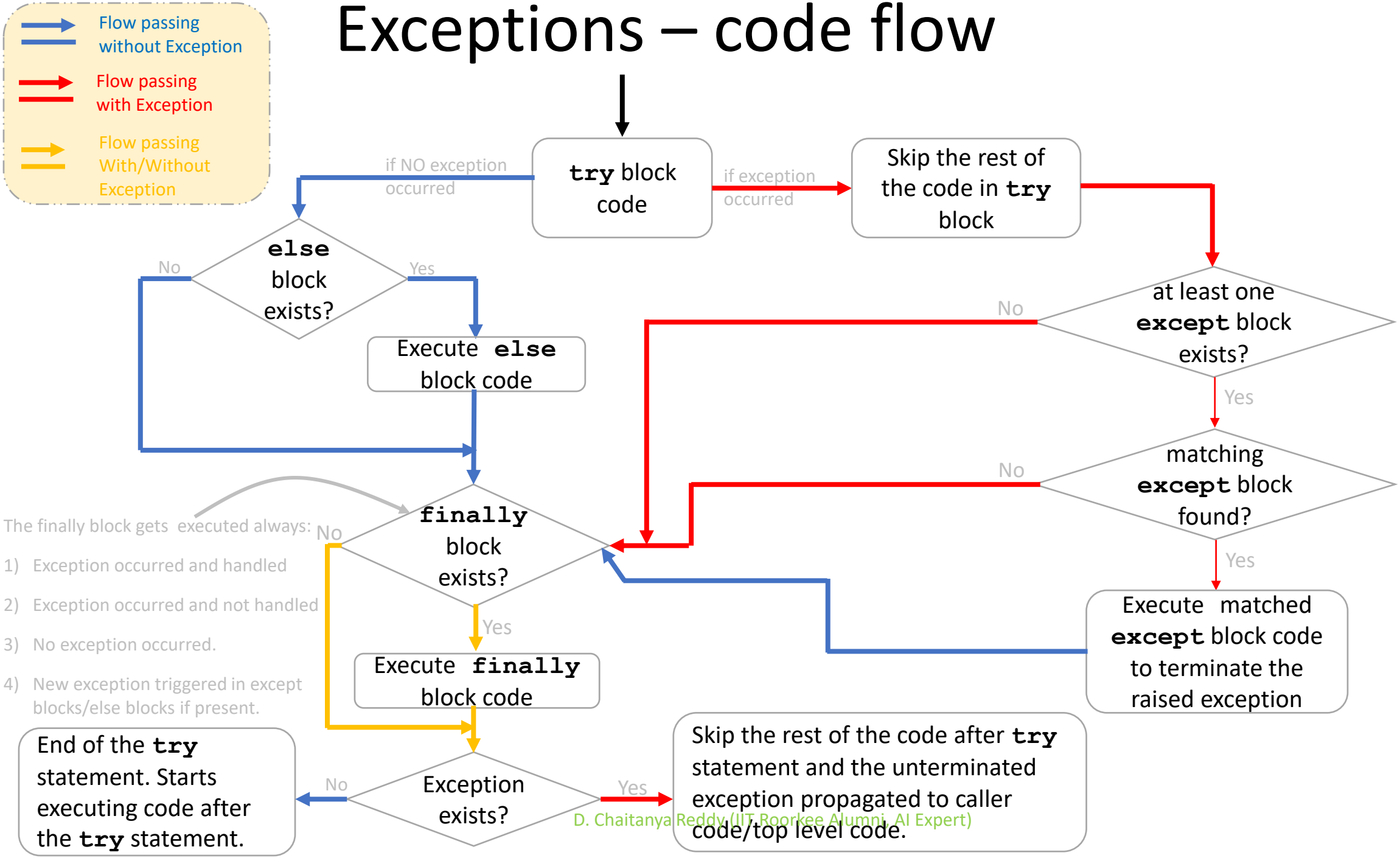
Try statement can be nested in

- **`try`** block
- all of the **`except`** block(s),
- **`else`** block,
- **`finally`** block

Possible-10

```
try:
    try: statement(s)
    except: statement(s)
except (name5, name6) as var:
    try: statement(s)
    except: statement(s)
    else: statement(s)
else:
    try: statement(s)
    except: statement(s)
    else: statement(s)
    finally: statement(s)
finally:
    try: statement(s)
    except: statement(s)
    finally: statement(s)
```

Exceptions – code flow



Exceptions – Examples

Ex1:

```
def action(x, y):
    print(x + y)


try:
    print('try block starting...')
    action([1, 2, 3], 6)
    print('try block ending...')
except NameError: print('got Name Error')
except (IndexError, KeyError):
    print('got Index or key error')
except AttributeError as var:
    print('got Attr Error:', var)
except (TypeError, ZeroDivisionError) as err:
    print('got Type or ZeroDivError.')
except: print('got some different error')
else: print('No Error found.')
finally:
    print('taking care of clean up code here')

print('After the try statement.')
```

```
Output:
try block starting...
got Type or ZeroDivError.
taking care of clean up code here
After the try statement.
```

Exceptions – Examples

Ex2:

```
def action(x, y):
    try:
        print('try block starting...')
        print(x + y)
        print('try block ending...')
    except NameError: print('got Name Error')
    except (IndexError, KeyError):
        print('got Index or key error')
    except AttributeError as var:
        print('got Attr Error:', var)
    except (OverflowError, ZeroDivisionError) as err:
        print('got Type or ZeroDivError.')
    else: print('No Error found.')
    finally:
        print('taking care of clean up code here')

    print('After the try statement.')

action([1, 2, 3], 6)
```

```
Output:
try block starting...
taking care of clean up code here
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [9], in <cell line: 4>()
      4 try:
      5     print('try block starting...')
----> 6     action([1, 2, 3], 6)
      7     print('try block ending...')
      8 except NameError: print('got Name Error')

Input In [9], in action(x, y)
      1 def action(x, y):
----> 2     print(x + y)

TypeError: can only concatenate list (not "int") to list
```

`except` block matching process

Ex2:

```
def action(x, y):
    try:
        print('try block starting...')
        print(x + y)
        print('try block ending...')
    except NameError:
        print('got Name Error')
    except AttributeError as var:
        print('got Attr Error:', var)
    except (IndexError, KeyError):
        print('got Index or key error')
    except (OverflowError, TypeError) as err:
        print('got Type or ZeroDivError.')
    finally: print('all clean up code here')
    print('After the try statement.')

action([1, 2, 3], 6)
```

1

This line in try block causes **`TypeError`** and reason for `TypeError` is **can only concatenate list (not "int") to list**

2

Create a new instance for `TypeError` class and pass that new instance to variable present in **`as`** clause in `except` block.

```
X = TypeError('can only concatenate list (not "int") to list')
raise X
```

exception class mentioned in `except` block must be either same as error instance's class or super class of error instance's class.

This is called **implicit exception raising**:

1) Python internally choose the right exception class to raise.
2) Python internally creates an instance for exception class.

```
Output:
try block starting...
got Type or ZeroDivError.
all clean up code here
After the try statement.
```

Exceptions – `raise` statement

```
def action(x, y):
   try:
      print(x + y)
   except (AttributeError) as err:
      print('got Type Error.')
   finally: print('all clean up code here')
   print('After the try statement.')


action('mahesh', 4)
```

```
Output:
all clean up code here
---------------------------------------------------
TypeError           Traceback (most recent call last)
Input In [12], in <cell line: 1>()
----> 1 action('mahesh', 4)

Input In [11], in action(x, y)
      1 def action(x, y):
      2    try:
----> 3       print(x + y)
      4    except (AttributeError) as err:
      5       print('got Type Error.')

TypeError: can only concatenate str (not "int") to str
```

This is implicit exception and we do not have any control on the message in implicit exceptions.

`raise` statement is used to raise exceptions **explicitly**.

```
def action(x, y):
   try:
      if isinstance(x, str):
         raise TypeError('Trying to add wrong types. please fix it.')
      else: print(x + y)
   except (AttributeError) as err:
      print('got Type Error.')
   finally: print('taking care of clean up code here')


action('mahesh', 4)
```

We explicitly should check the potential failure condition

Should explicitly raise the exception.

Should explicitly give some more meaningful error messasge

```
Output:
taking care of clean up code here
---------------------------------------------------------------
TypeError                          Traceback (most recent call last)
Input In [14], in <cell line: 1>()
----> 1 action('mahesh', 4)

Input In [13], in action(x, y)
      1 def action(x, y):
      2    try:
----> 3       if isinstance(x, str): raise TypeError('you are tyring to
add wrong types. please fix it.')
      4       else: print(x + y)
      5    except (AttributeError) as err:
TypeError: you are tyring to add wrong types. please fix it.
```

This exception raised explicitly and hence have full control on the message.

Exceptions – `raise` statement

```
class MyExc(Exception): pass

def action(x, y):
   try:
      if isinstance(x, str):
         raise MyExc('Error found. Fix it.')
      else: print(x + y)
   except (AttributeError) as err:
      print('got Type Error.')
   finally: print('taking care of clean up code here')


action('mahesh', 4)
```

We can raise exception belongs to any exception class.

```
Output:
taking care of clean up code here
---------------------------------------------------------------
MyExc                            Traceback (most recent call last)
Input In [18], in <cell line: 11>()
      8       print('got Type Error.')
      9    finally: print('taking care of clean up code here')
---> 11 action('mahesh', 4)

Input In [18], in action(x, y)
      3 try:
      4    if isinstance(x, str):
----> 5       raise MyExc(Error found. Fix it.')
      6    else: print(x + y)
      7 except (AttributeError) as err:

MyExc: Error found. Fix it.
```

This exception raised explicitly and hence have full control on the type of exception too.

`raise` statement forms

```
class MyExc(Exception): pass


def action(x, y):
   try:
      if isinstance(x, str):
         raise MyExc(Error found. Fix it.')
      else: print(x + y)
   except MyExc as err:
      print('got Type Error.')
   finally: print('all clean up code here')


action('mahesh', 4)
```

Form-1: `raise` statement followed by instance of some exception class.

Here, instance is created within the `raise` statement.

```
Output:
got Type Error.
all clean up code here
```

```
class MyExc(Exception): pass


def action(x, y):
   try:
      if isinstance(x, str):
         exc = MyExc(Error found. Fix it.')
         raise exc
      else: print(x + y)
   except MyExc as err:
      print('got Type Error.')
   finally: print('all clean up code here')


action('mahesh', 4)
```

Form-1 (alternate): `raise` statement followed by instance of some exception class.

Here, instance is created before the `raise` statement.

```
Output:
got Type Error.
all clean up code here
```

`raise` statement forms

```
def action(x, y):
   try:
      if isinstance(x, str):
         raise TypeError
      else: print(x + y)
   except AttributeError as err:
      print('got Attribute Error.')
   finally: print('all clean up code here')
action('mahesh', 4)
```

Form-2: `raise` statement followed by name of some exception class.

Here, No instance for any exception class gets created.

However, python creates instance for the exception class with empty error message.

```
Output:
all clean up code here
---------------------------------------------------
TypeError          Traceback (most recent call last)
Input In [25], in <cell line: 10>()
      7       print('got Attribute Error.')
      8    finally: print('all clean up code here')
---> 10 action('mahesh', 4)

Input In [25], in action(x, y)
      2 try:
      3    if isinstance(x, str):
----> 4       raise TypeError
      5    else: print(x + y)
      6 except AttributeError as err:

TypeError:
```

```
def action(x, y):
   try:
      if isinstance(x, str):
         raise TypeError('spam')
      else: print(x + y)
   except TypeError as err:
      print('propagating Type Error')
      raise
action('mahesh', 4)
```

Form-3: simple `raise` statement.

It will re raise the most recent exception

```
Output:
propagating Type Error
---------------------------------------------------------
TypeError            Traceback (most recent call last)
Input In [26], in <cell line: 12>()
      9       print('propagating Type Error')
     10       raise
---> 12 action('mahesh', 4)

Input In [26], in action(x, y)
      4 try:
      5    if isinstance(x, str):
----> 6       raise TypeError('spam')
      7    else: print(x + y)
      8 except TypeError as err:

TypeError: spam
```

`raise` statement – multiple args

Multiple values can be passed while raising exception.

```
class MyExc(Exception): pass
def action(x, y):
    try:
        if x < 0: raise MyExc('negative number', x, 'addition')
        else: print(x + y)
    except MyExc as ex:
        print(ex.args)
        print(ex.args[0], ex.args[1])


action(-2, 4)
```

Custom/own Exception class

All these multiple values are saved in `args` attribute of error instance as `tuple`.

```
Output:
('negative number', -2, 'addition')
negative number -2
```

```
def action(x, y):
    try:
        if isinstance(x, str):
            raise TypeError('not a number', x, 'addition')
        else: print(x + y)
    except Exception as ex:
        print(ex.args)
        print(ex.args[1], ex.args[2])


action('mahesh', 4)
```

Built-in Exception class

All these multiple values are saved in `args` attribute of error instance as `tuple`.

```
Output:
('not a number', 'mahesh', 'addition')
mahesh addition
```

`raise from` statement

```
class MyExc(Exception): pass


def action(x, y):
   try:
      print(x + y)
   except TypeError as err:
      print('propagating Type Error.')
      raise MyExc('Bad Types') from err
   finally: print('all clean up code here')


action('mahesh', 4)
```

Explicit Exception Chaining

Raising a new exception from another exception which is called **exception chaining.**

Since, we are explicitly raising new exception from another exception, this is called **explicit exception chaining**

```
Output:
propagating Type Error.
all clean up code here
---------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [28], in action(x, y)
      4 try:
----> 5    print(x + y)
      6 except TypeError as err:

TypeError: can only concatenate str (not "int") to str

The above exception was the direct cause of the following exception:

MyExc                                     Traceback (most recent call last)
Input In [28], in <cell line: 11>()
      8       raise MyExc('Bad Types') from err
      9    finally: print('all clean up code here')
---> 11 action('mahesh', 4)

Input In [28], in action(x, y)
      6 except TypeError as err:
      7    print('propagating Type Error.')
----> 8    raise MyExc('Bad Types') from err
      9 finally: print('all clean up code here')

MyExc: Bad Types
```

Implicit Exception Chaining

```
def action(x, y):
    try:
        print(x + y)
    except TypeError as err:
        print('propagating Type Error.')
        cnt = len(x)
        cnt -= 6
        res = 100/cnt
    finally: print('all clean up code here')


action('mahesh', 4)
```

Implicit Exception Chaining

Python automatically, raises a new exception while terminating the previous exception.

Since, python automatically raising new exception while handling another exception, this is called **implicit exception chaining**

```
propagating Type Error.
all clean up code here
---------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [29], in action(x, y)
      2 try:
----> 3     print(x + y)
      4 except TypeError as err:

TypeError: can only concatenate str (not "int") to str

During handling of the above exception, another exception occurred:

ZeroDivisionError                         Traceback (most recent call last)
Input In [29], in <cell line: 11>()
      8         res = 100/cnt
      9     finally: print('all clean up code here')
---> 11 action('mahesh', 4)

Input In [29], in action(x, y)
      6     cnt = len(x)
      7     cnt -= 6
----> 8     res = 100/cnt
      9 finally: print('all clean up code here')

ZeroDivisionError: division by zero
```

`assert` statement

`assert` statement is nothing but **conditional `raise`** statement.

```
class MyExc(Exception): pass

def action(x, y):

   try:
      if x < 0: raise MyExc('negative number')
      else: print(x + y)
   except TypeError as err:
      print('propagating Type Error')

action(-2, 4)
```

This is not a programming error. Raising exception based on user defined constraint.

Output:

```
---------------------------------------------------
MyExc                     Traceback (most recent call last)
Input In [7], in <cell line: 9>()
      6    except TypeError as err:
      7       print('propagating Type Error')
----> 9 action(-2, 4)

Input In [7], in action(x, y)
      2 def action(x, y):
      3    try:
----> 4       if x < 0: raise MyExc('negative number')
      5       else: print(x + y)
      6    except TypeError as err:

MyExc: negative number
```

`assert` is mostly intended for **trapping user-defined constraints**, not for catching genuine programming errors.

Syntax: `assert test, data` — `data` is an optional

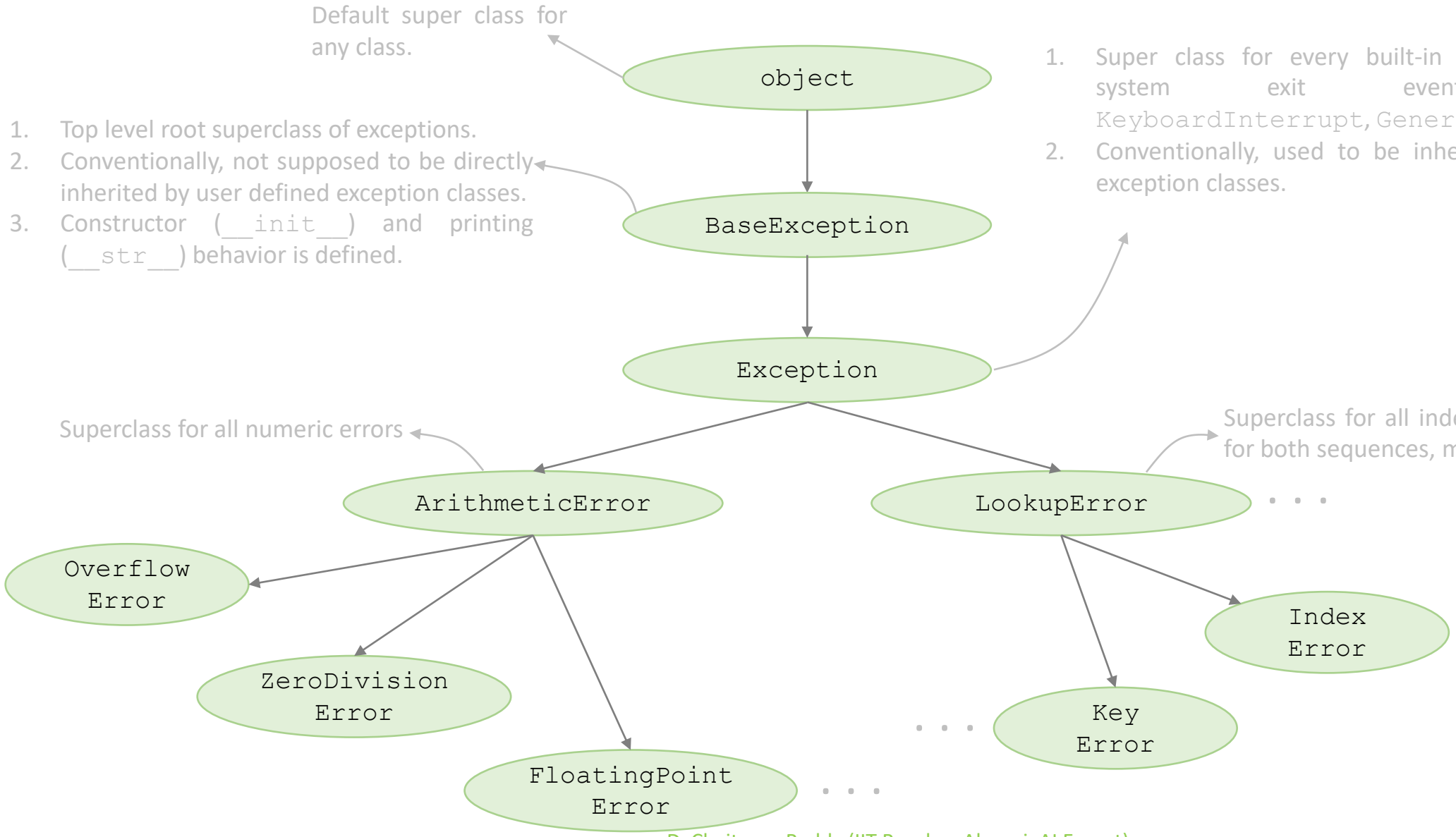
```
def action(x, y):

   try:
      assert x>0, 'negative number'
      print(x + y)
   except TypeError as err:
      print('propagating Type Error')

action(-2, 4)
```

1) Evaluates the **`test`** condition `(x>0)`.
2) If **`False`**, then raises always **`AssertionError`**.

Output:

```
---------------------------------------------------
AssertionError     Traceback (most recent call last)
Input In [14], in <cell line: 8>()
      6       print('propagating Type Error')
----> 8 action(-2, 4)

Input In [14], in action(x, y)
      1 def action(x, y):
      2    try:
----> 3       assert x>0, 'negative number'
      4       print(x + y)
      5    except TypeError as err:

AssertionError: negative number
```

Built-in Exception Classes



E N D