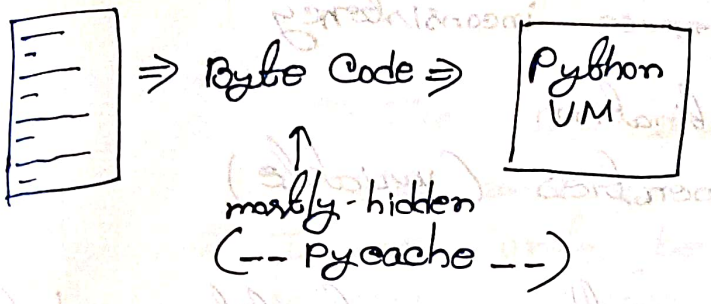
Python

Tamal Mallick

Pythons inner working

python main.py



1. compile to Byte code (low level & platform Independ

→ Byte code runs faster

.pyc → compiled python (Frozen Binaries)

--Pycache-- (-- xyz --) → reflects useful for internal working of python

Source change & Python version

main.cpython-313.pyc

→ Works only for imported files

→ Not for top level files

Python Virtual Machine (PVM)

→ Code loop to iterate byte code

→ Run time engine

→ Also known as python interpreter

Byte code is not machine code

→ python specific interpretation

→ cpython, jPython, PyPy, IronPython, stackless

(standard) (deals with java's binary) (performance) (concurrency)

Python shell

NameError → no variable/def of that name

IndentationError → Tab / space inconsistency

os.getcwd() → destination

sys.platform → properties (variable)

When we import python import file as bytecode that's why __pycache__ get created.

So to fetch new changes of imported file ~~we can re-import it use~~ the file must be converted to bytecode only after that the new changes will be reflected.

→ close shell & re-import

→ (or) from importlib import reload

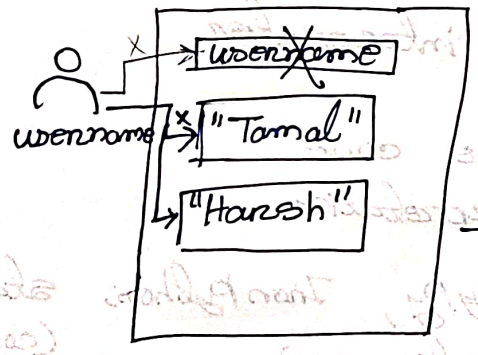
reload (hellow)

Now new changes are useable

Immutable & Mutable

username = "Tamal", username = "Harsh"

x = 10
y = x
x = 15
Print(x, y)
(15, 10)



→ In python instead of referring variable we refer value of the variable as a object.

→ First username = 'Tamal' means reference of username is object holding string "Tamal"

→ username = "Harsh" reference of username is object "Harsh"

→ Now no variable (label or pointer that refers to the actual object) refers object 'Tamal' so it is eligable for garbage collection.

reference counting
how many references there are to an object.
it o eligible for garbage collector

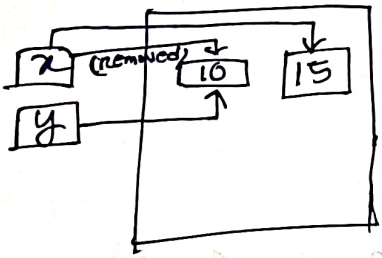
Variables are reference or label to objects.

Immutable :→ Value of an object that has been created inside the memory can't not be changed. (Only garbage collector deallocates the memory when its reference count=0)

or. The value of an object created in memory can't be changed after creation. It can only be deallocated by python's garbage collector when there are no reference to it.

Modification creates new object and var. refers to that instead of changing object's value.

x=10
y=x
x→10
y→10
x=15
x→15
y→10



Mutable → Value of an object is changable without creating any new object in memory. The original object is modified.

Variable stores the reference
Object stores the value/values } Non-primitive
python, JS,

Primitive : Variable holds actual value directly in memory. (ex. C, C++, java, assembly stores int, float etc directly)

Python supports high level abstractions (like obj, collection etc) and allows more complex programming that's why its high level language.

C, C++ deals with simple datatypes char, int, float & allows direct memory manipulation revolves around these basics

and control over machines hardware thats why these are low level languages

Python Data Types

Number

String

Lists

Tuple

[Bracket] (Parentheses) {Braces}

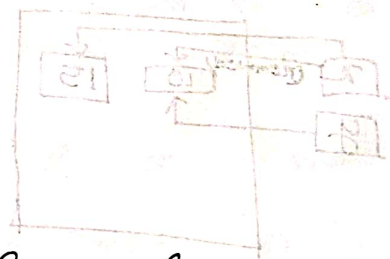
Dictionary

Set

File

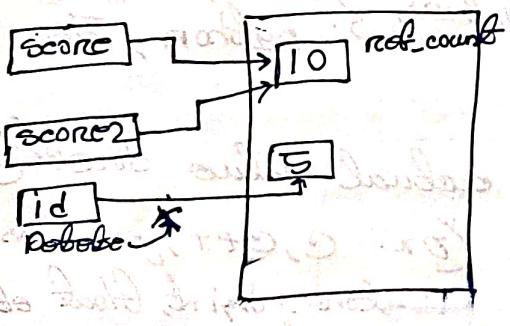
None

Function, module, class



Advanced: Decorators, Generators, Iterators, Meta Programming

Internal Working
(Copy, reference counts, slice)



Python compiler use reference count, Though getting the exact number is tricky as python is not so close to the memory thats why sys.getrefcount() repeat some specific number due to internal compiler optimization loop, return same number almost each time.

Python store data & its type in memory, variables has no connection with the data type.

Reference in memory has both data & data type, thats why we get error in invalid operation

In python variables have no data type but each reference inside memory has its own.

Python treats numbers & strings seperately

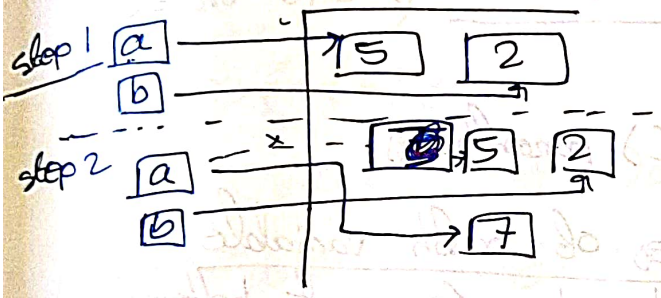
Python garbage collection ~~free up~~ unlike other data types clear those (num, str) often certain daily to optimize and avoid assign-re-assignment of those references to reuse them & save computation

Var	Ref obj
a=3	(3)a
a=3.5	(3.5)a (3)
b=2, a=5	(2)b (5)a (3) (3.5)

```
a = 5
b = 2
a = a+2
=> a = 7
```

```
a = a+2
=> a = 5+2   → CPU/GPU
=> a = 7     ← 7
```

a is referring the new object 7 ← 5 is eligible for G.C.

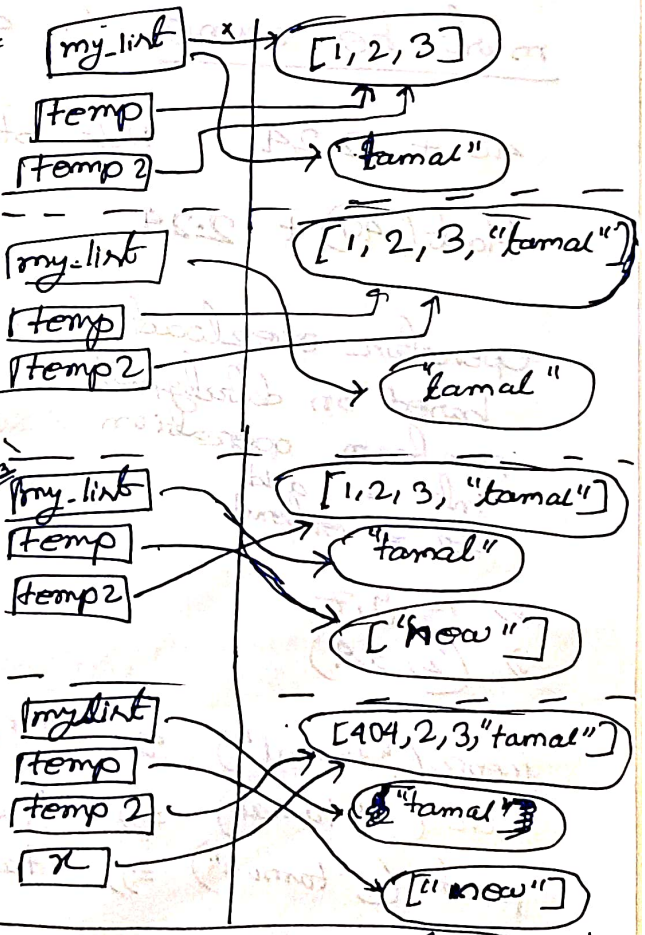


s1
```
my_list = [1, 2, 3]
temp = my_list
my_list = "tamal"
```
→ immutable so new obj created

s2
```
temp2 = temp
temp.append("tamal")
```

s3
```
temp = ["new"]
```
(as assigning new object instead of changing or changing the existing one)

s4
```
x = temp2
x[0] = 404
```



copy: h[:], n.copy, copy.copy(h), copy.deepcopy(h) → creates new obj

```
a = b = [1, 2, 3]
a == b
a is b
```
|> True (checks if values are same)
|>> True (checks if both referring to same object in memory; same mem loc)

```
a = b[:]
a == b
a is b
```
|> True (Both has same values)
|>> False (Both refers different objects in memory, diff mem location)

Numbers in Depth

Python numbers : bool < int < float < complex

Decimal , Fraction, Arbitary precision int

↓

using int we can store int of 4300 digits max by default

Though using sys.set_int_max_str_digits()

Keep No ambiguity: avoid operation precedence (>640 or =0)

x+y * z Not standard

(x+y) * z Standard using () parenthesis

In calculation datatypes of both variable must be same : Standard (convert before calculation)

40 + 2.24 Not Standard

float(40) + 2.24

Operation overloading : based on datatypes perform operation

int = add

str = embend

"Tamal_" + "Mallick" = 'Tamal_Mallick'

power = a ** b

Mod = a % b

⇒ x, y, z

⇒ (2, 3, 4)

repr("tamal") ⇒ "'tamal'"

str("tamal") ⇒ 'tamal'

print("tamal") ⇒ tamal

In python True & False are treated as numbers internally (1 & 0), using repr showed as True or False

x < y < z → x < y and y < z

1 == 2 < 3 → (1 == 2) and (2 < 3) → False and True
↓
False

math.floor(2.4) ⇒ 2
math.floor(-2.4) ⇒ -3
} closest int value ≤ the number

math.trunc(2.4) ⇒ 2
math.trunc(-2.4) ⇒ -2
} closest int near 0

python calculation is highly precise
supports imaginary numbers i = √-1, j = √-1

a = 2 + 3j
a = a * 2
a ⇒ 4 + 6j

Octal 0o XXX... eg: 0o 25

Binary 0b XXX... eg: 0b 101

Hex 0x XXX... eg: 0x 64

Convert Decimal to others
int ('64', 8)
int ('64', 16)
int ('64', 2)

Bit shifting
x << 2 Left shift by 2 bits
x >> n Right shift by 2 bits

Random
random.random() random.randint(0,10) random.choice(['a','b','c'])
random.shuffle(['a','b','c'])

0.1 + 0.1 + 0.4 = 0.600....01
0.1 + 0.1 + 0.1 = 0.300...09
0.1 * 3 = 0.300...04
(0.1 + 0.1 + 0.1) - 0.3 == 0 ? Ans = False
Thats why we use Decimal library to handle decimal accuracy
Decimal('0.1') + Decimal('0.1') + Decimal('0.1')
= Decimal('0.3')

Set

{1,3} & {1,2,3,4} ⇒ {1,3} → set intersection (and)

{1,3} | {2,4} ⇒ {1,2,3,4} → set union (or)

{1,3} - {3,4} ⇒ {1}

{1} - {1} ⇒ set() as {} treated as dictionary

True == 1 ⇒ True

False == 0 ⇒ True

True is 1 ⇒ False Warning as 'is' is use to compare reference or object's location in memory

True + 5 ⇒ 6

String in Python

num = "0123456789"

num [start : end : step] slicing

if step > 0 then start ≤ end

if step < 0 then start ≥ end

python string supports different unicode characters

s = " xyz " s.strip() = "xyz"

s.replace("xyz", "tamal") = " tamal "

s = "1,2,3" s.split(",") = ['1','2','3']

s.find(",1") = 0

s.find("12") = -1

chai = "Mint chai chai chaichai"

chai.count("chai") = 4

Place holders

order = "I ordered {} cups of {} chai"

order.format(3, "Masala") ⇒

I ordered 3 cups of Masala chai"

a = ["1", '2', '3', '4'] ', '.join(a) ⇒ "1, 2, 3, 4"

"He said, \"I am okay\"."

⇒ "He said, "I am okay".

print ("A\tB") ⇒ A B

print (r"A\tB") ⇒ A\tB

Like windows path problem due to \

as \ is used for unicode character

(unicode error)

List in Python

nums = ['0','1','2','3','4']

print(nums[1:2]) ⇒ '1'

nums [1:2] = 'abc'

print(nums) ⇒ ['0','a','b','c','2','3','4']

∴ 'abc' treated each char as array item.

nums = [0,1, 2,3,4]

nums [1:2] = ['abc']

print (nums) ⇒ [0, 'abc', 2, 3, 4]

nums [1:3] = ['Tamal', 'Mallick']

print (nums) ⇒ [0, 'Tamal', 'Mallick', 3, 4]

nums = [0,1,2,3]

nums [1:] = [404, 404]

nums ⇒ [0, 404, 404, 2, 3]

nums [1:] = [] # Inserted nothing at those position hence deleted.

nums ⇒ [0,1,2,3]

Dictionary in Python

dict[ ] ⇒ value or keyError

dict.get( ) ⇒ value or None

Mutable, popitem() removes last entry

subbonbs comprehension {x:x*x for x in range(1,5)}

Newdict = dict.fromkeys(['A','B','C'], 'ABC')

= {'A': 'ABC', 'B': 'ABC', 'C': 'ABC'}

Tuple in python

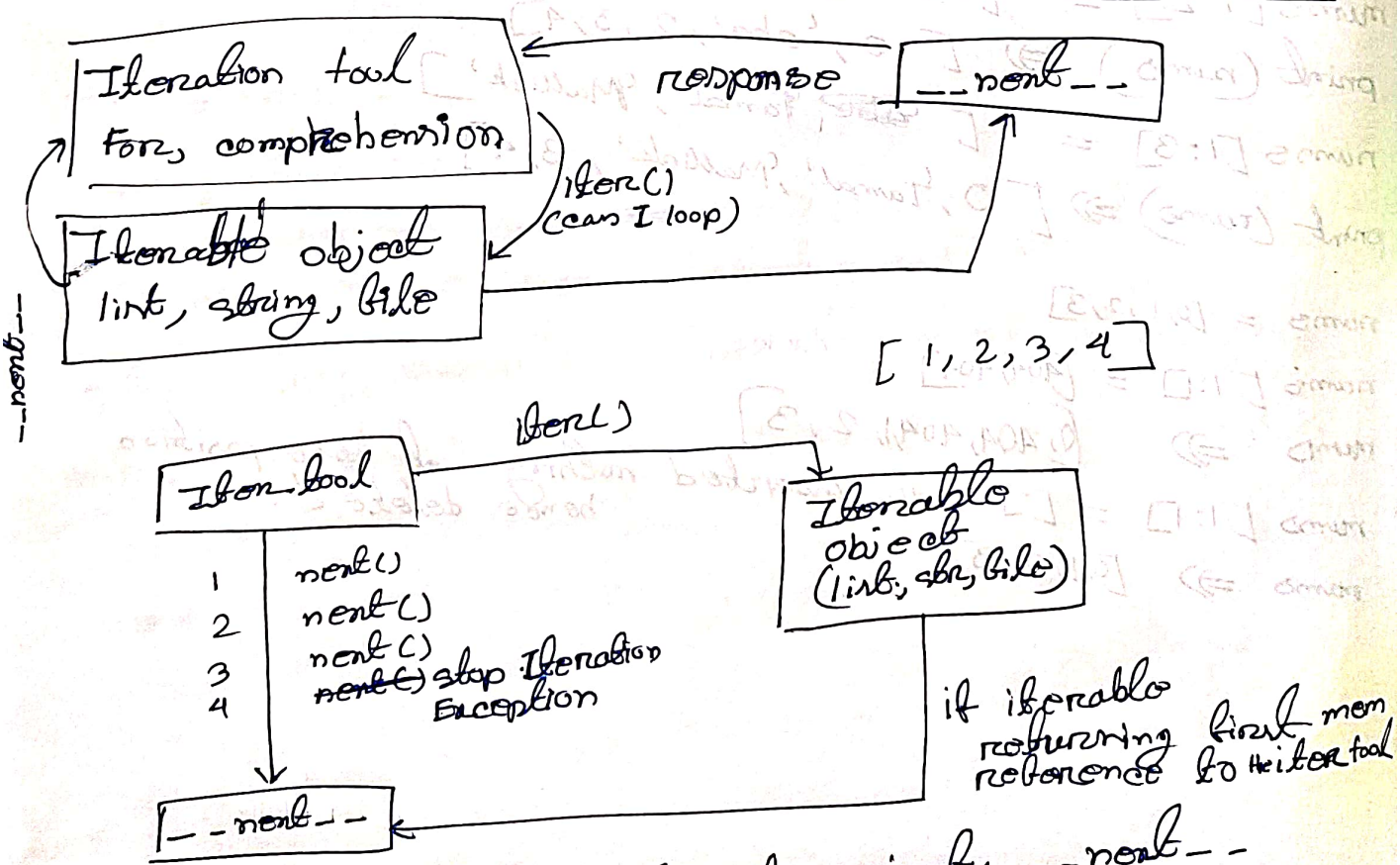
Immutable

supports all remaining operations like list()

a = (1,2,3,1,4)

a.count(1) ⇒ 2

Behind the scene of Loops in Python



If a object has internal property __next__ then only it is iterable.

```
>> f = open('file.txt')
>> f.__next__()
Tamal Mallick
>> f.__next__()
Hellow
>> f.__next__()
Last line
>> f.__next__()
StopIteration
f.__next__()
```

Tamal Mallick
Hellow
Last line
file.txt

The content ID over

Memory reference of list iteration always points to the first, doesn't change with iteration.

Functions in Python

1. Basic

```
def square_of_num(parameters):        # parameter variable
    return parameters ** 2
def fun(x=2):                         # 2. Default parameter value (x=2)
    print(x)
```

3. Lambda function

```
cube = lambda x: x**3
print(cube(3))
```

the function object stored in cube variable without any internal name. No __name__ attribute for this object.

For cube.__name__ = '<lambda>'

where fun.__name__ = 'fun'

4 Function with *args

Takes any n number of arguments during function call

5. Function with **kwargs

Takes n number of keyword arguments as a dictionary.

6. Generator Function with yield.

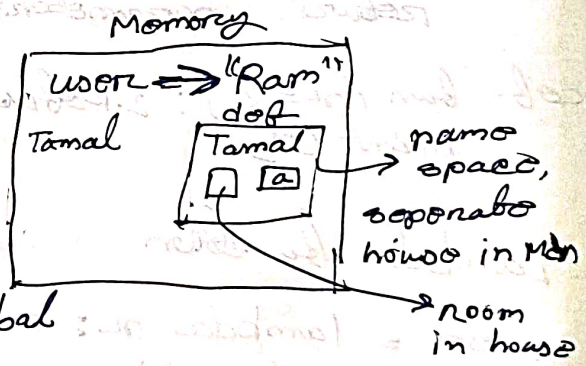
~~Generator~~ Returns object having code and current state that produce & return value on demand instead of storing all iterables in the memory at the beginning.

Scopes and Closure

```
x = 99
def fun():
    global x
    x = 12

fun()
print(x) >> 12
```

global → refer x from global memory scope




```
def f1():
    x = 88
    def f2(): print(x)
    return f2

y = f1()
y()
>> 88
```

→ Here f1 returns definition f2 and all the references associated with it

```
def f(a):
    def actual(b): return b**a
    return actual
g, h = f(2), f(3)
```

g storing a=2 as its bag pack
h ← a=3 for h as a closure

g(3) >> 8
h(3) >> 27

closure allows to store accable values from parent and use those stored values in its subfunctions (in Nested function).

global keyword provides read/write permission to any variable only residing in the global scope. After using it we access/update object having memory reference in the global scope.

By default any child has read only access of its parent of any level. (Upper level)

```
var = "Global"
def fun():
    var = "local"
    def child():
        global var
        var = 'child'
    print(var)
    child()
    print(var)
fun()
print(var)
```

>> local
>> local
>> child

Object Oriented Programming

OPP is like creating a metal casting mold -- the class is the mold, designed with code (Python, C++, Java) to define structure and features. When we create an object -- it's like making a doll from the mold allocating memory, applying the class design and getting fully shape bases.

Like customizing the doll with hair and costumes -- similar to calling methods on the object with specific parameters.

Each doll is unique self sufficient -- Each object is also similar. Mold is reusable -- class is also reusable and produce as many object we need.

OPPs helps us to build modular, reusable and organized code -- just like a factory producing various custom dolls from same reliable mold.

Also like generalized bank's form

Form pdf → Many A4 copy → People fill it up & use

Decorators in Python

Decorators replaces the original function's reference with a new wrapped object in memory so every future call executes the wrapper object (script, original function, script), not the original function directly.

@decorator
say_hi():.. Mem reference associated with function identifier in namespace "say_hi" has been changed by @decorator.
..... New reference is the reference of wrapper.

```
def decorator(function):
    def wrapper(*args, **kwargs):
        ..... 
        res = function(*args, **kwargs)
        .....
        return wrapper res
    return wrapper   # new reference of fun identifier
```