4/6/25

Python Basics

Topics to cover:
- ✓ Data Types
- ✓ Functions
- ✓ Decorators
- ✓ Loops
- ✓ Lists
- ✓ Tuples

DATA TYPES

Data types are means to identify type of data and set of valid operations for it. Python offers built-in core data types like:

1) Integers → Signed (int)
   → Booleans (0,1)
2) Float
3) Complex → real and imaginary part.
4) Strings
5) List
6) Tuple
7) Dictionaries
8) Sets

User defined data types in Python allows programmers to create custom data structures tailored to specific needs, extending beyond the built-in data types. These custom types are primarily implemented using classes.

Mutable and Immutable Types

↓	↓
modifiable	non-modifiable
lists, sets	integers, float
dictionaries	Booleans, string, tuples

Q

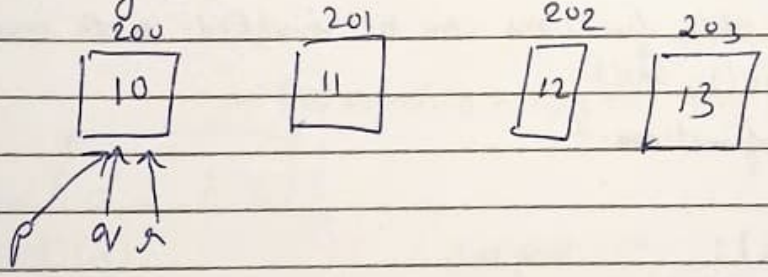
On what basis do we decide which data type is mutable or immutable?

Ans

In the memory, if we do the following:

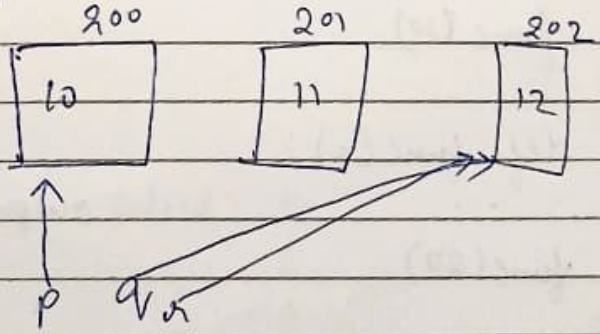
```
p = 10
qv = p
r = 10
```

The memory is



but as soon as we implement this code:

```
p = 10
qv = 12
r = qv
```



They were made to point to different integer objects.

In immutable data types, we can change the values in the same memory address. Eg:

Let us take a list d=[1,2,3,4,5]. if we want to make a change, we can easily do so by ~~iterating~~ indexing it

```
d=[1,2,3,4,5]
d[2] = 10
```

Now the value of d has changed to [1,2,10,4,5].

FUNCTIONS

A function is a subprogram that acts on data and often returns a value.

Types of functions:

-> Built-in functions : len(), type(), int(), input() etc..

-> functions defined in modules - when a module needs to be imported and then functions can be invoked. Math module has pow(), sin(), sqrt().

-> User defined functions.

```
def func(a):
```

output: Indentation error

```
func(10)
```

```
def func(a):
    ...
func(20)
```

output : ____ (no output but no error)

```
def func(*a):
    print(a)
    print(type(a))
func(1,2,3,4,5)
```

output: (1, 2, 3, 4, 5)
<class 'tuple'>

```
def func(*a,b):
    print(a)
    print(b)
func(1,2,3,4,5)
```

output: Type Error

Q Why?

Ans: Any parameters defined after the *a are treated as keyword-only argum and not as positional arguments.

```
def func(*a, b=10):
    print(a)
    print(b)
func(1,2,3,4,5)
```

output: (1, 2, 3, 4, 5) {tuple}
10 {int}

→ Parameters - they are only place holders

```
def func(a=2, b=4):
    print(a)
    print(b)
func(10, 20)
```

→ arguments

output 10
20

```
def func(a, b, *, c=3, d=4, e):
    print(a)
    print(b)
    print(c)
    print(d)
    print(e)
func(1,2)
```

output: 1
2
8
4
9

Q- difference between *a and **a

Ans

When we have *a, we can give arguments as follows:

```
def func(*a):
    print(a)
func(1,2,3,4,5)
```

*a → variable positional args

when we use **a, we give arguments like:

```
def func(**a):
    print(a)
func(x=7, y=8, z=9)
```

**a → variable keyword args

* The output would be {'x': 7, 'y': 8, 'z': 9}. **a returns a dictionary.

DECORATORS

```
def decorator(my_function):
    def inner():
        print("greetings")
        my_function()
    return inner

def my_function():
    print("Hello")

a = decorator(my function)    {inner()}
print(a)
```

```
def a(func):                    | func(*p, **q)
    def inner(*p, **q):         | func(*p, **q)
        print(f"called with {*p} {**q}")
        return func(*p, **q)
    return inner
```

(3,4)

```
@a
def add(a, b):
    return {a+b}
print(add(3,4))
```

Output: called with (3,4) {}
7

LOOPS

```
for loop
    d = [1, 2, 3]
    for i in d:
        print(i)
```

output:
1
2
3

```
string = "python"
for char in string:
    print(char)
```

```
for j in range(2, 10, 2):
    print(j)
```

2
4
6
8

While loop

```
i=0
while i<5:
    print(i)
    i+=1
```

Control statements

break - exit the loop immediately

Continue - skip to the next iteration

pass - do nothing

```
for i in range(5):
    print(i)
else:
    print("loop finished successfully")
```

else → runs only when the for loop didn't end with break

using enumerate()

```
animals = ["dog", "cat", "horse", "donkey"]
for index, animals in enumerate(animals):
    print(index, animals)
```

output:

```
0 dog
1 cat
2 horse
3 donkey
```

looping with zip() combines two iterables into a single tuple

```
names = ["bob", "Harry"]
marks = [85, 90]

for name, marks in zip(names, marks):
    print(f"{name} scored {marks}")
```

```
names = ["Bob", "Harry"]
marks = (85, 90)

for name, marks in zip(names, marks):
    print(f"{name} scored {mark}")
```

Both output is same

Bob scored 85
Harry scored 90

List Comprehension

```
Cube = [x*x*x for x in range(5)]
print(cube)
```

[0, 1, 8, 27, 64]

```
even = [x for x in range(8) if x%2 == 0]
print(even)
```

[0, 2, 4, 6]

LISTS

l = list() → generates an empty list and names it l
l = list("hello")
l: ["h", "e", "l", "l", "o"]

Joining lists

Concatenation operator → * +

l1 = [1, 2, 3]
l2 = [4, 5, 6]
l1 + l2 = [1, 2, 3, 4, 5, 6]

Replication operator → *

[1, 4, 9] * 3 → [1, 4, 9, 1, 4, 9, 1, 4, 9]

Slicing the list

l = [1, 2, 3, 4, 5, 6] [start : stop : step]
l[::-1] → reverse
l[2:30] → [3, 4, 5, 6]
l[-20:3] → [1, 2, 3]

l = ["one", "two", "three"]
l[0:2] = "a"
» ["a", "three"]

List Manipulation

```
l1 = [10, 20]
l1.append(16)
» [10, 20, 16]
```

del l1[0:1] → deletes all the elements between 0 and 1
» [20, 16]

l1.index(20) → 0

l2 = [15, 14, 13]

l1.extend(l2) → [20, 16, 15, 14, 13]

* l2 remains unchanged.

append adds only 1 element at a time but extend adds more than one.

l1.insert(1, 15) → [20, 15, 16]
(1 → index, 15 → element)

append does not return a new list. It just modifies in-place.

pop()

list.pop(<index>) returns the element being deleted
(<index> → optional)

~~[illegible]~~ → also works for None, int.

remove() will remove the first instance of given item from list

```
l = ["a", "a", "b", "c"]
l.remove("a")
```

↳ does not return anything.

```
l = ['a', 'b', 'c']
```

Q- clear() → clears the whole list and makes it empty

```
l1 = [1, 2, 3]
l1.clear()
l1 = []
```

* del even deletes the list and frees the space, but clear just removes the elements and keeps the variable in the memory.

Q- if clear() only removes the elements in the list, why does the memory address change before and after using the .clear().

↳ X, id doesn't change. Wrong question.

count() → counts the number of times the element passed as arguments exists in the list

```
l = [None, 1, None]
l.count(None) → 2
```

~~l = [~~

reverse() → reverses the items of the list does not create a new list or return anything

```
list.reverse()
```
↳ takes no arguments

sort() method → sorts items in the list. by default - ascending. done in-place.

```
list.sort() → ascending order
list.sort(reverse=True) → decreasing order.
```

TUPLES

Immutable sequences

tuple()

```
l = [1,2,3]
tuple(l) → (1,2,3)
```

tuple("string") → ("s", "t", "r", "i", "n", "g")

dict = {'a': 'apple', 'b': 'ball'}
tuple(dict) → ('a', 'b')

len, count, indexing and slicing, concatenation and replication same as lists.

Unpacking tuples

a, b, c, d = (1, 2, 3, 4)
a → 1
b → 2
c → 3
d → 4

a = [1, 2, 3]
b = a # deep copy
b.append(4)
print(a) → [1, 2, 3, 4]

a = [1, 2, 3]
b = a[:] # shallow copy.
b.append(4)
print(b) → [1, 2, 3]

a = [1] * 3 [1, 1, 1]
a[0] += 5
print(a) → [6, 1, 1]

```
def g(a, b=[]):
    b.append(a)
    return b
print(g(1))          [1]
print(g(2))          [1,2]
print(g(3,[]))       [3]
```

```
def add(*a):
    return sum(a)

print(add(1,2,3,4))      10
print(add())             0
```

```
def func(x, *args):
    return x + sum(args)

print(func(1,2,3,4))
print(func())            error - needs an argument.
```

```
def func(**a):
    for k,v in a.items():
        print(f"{k}:{v}")

print(fruit="mango", colour="yellow")
```

fruit = "Mango"
colour = "Yellow"
↳ both are keys

Keyword arg
positional arg

```
def hello( name , **b ):
    print("Hello", name )
    print( b)
```

Hello bob
{age:30, city:NY}

hello("bob", 'age': 30 , city = 'NY')