

Python Fundamentals

day 1

Today's Agenda

- What is python?
- Why python is so famous?
- History of python
- Comparison with C, JAVA
- ALL, MLL, HLL
- Compiler vs Interpreter



What is python??

Python is an interpreted, high-level, general-purpose programming language. Created by Guido van Rossum and first released in 1991, Python's design philosophy emphasizes code readability with its notable use of significant whitespace. Its language constructs and object-oriented approach aim to help programmers write clear, logical code for small and large-scale projects.

It supports multiple programming paradigms, including structured (particularly, procedural), object-oriented, and functional programming.

It is used for:

- web development (server-side),
- software development,
- mathematics,
- system scripting.

Why python is so famous and most recognized language?

First and foremost reason why Python is much popular because it is highly productive as compared to other programming languages like C++ and Java. It is much more concise and expressive language and requires less time, effort, and lines of code to perform the same operations.



The Python features like one-liners and dynamic type system allow developers to write very fewer lines of code for tasks that require more lines of code in other languages. This makes Python very easy-to-learn programming language even for beginners and newbies. For instance, Python programs are slower than Java, but they also take very less time to develop, as Python codes are 3 to 5 times shorter than Java codes.

Python is also very famous for its simple programming syntax, code readability and English-like commands that make coding in Python lot easier and efficient.

History of Python

Python was conceived in the late 1980s by Guido van Rossum at Centrum Wiskunde & Informatica (CWI) in the Netherlands as a successor to the ABC language (itself inspired by SETL), capable of exception handling and interfacing with the Amoeba operating system.



Python's name is derived from the British comedy group Monty Python, whom Python creator Guido van Rossum enjoyed while developing the language. Monty Python references appear frequently in Python code and culture; for example, the metasyntactic variables often used in Python literature are *spam* and *eggs* instead of the traditional *foo* and *bar*. The official Python documentation also contains various references to Monty Python routines.

Comparison with C and JAVA

Let us consider an example of swapping of two numbers in all top three programming languages.

Let's SWAP it

Case 1 - C language

```
#include <stdio.h>

void main()
{
    int a = 10;
    int b = 20;
    int temp;

    printf("a = %d",a);
    printf("b = %d",b);
    temp = a;
    a = b;
    b = temp;
    printf("a = %d",a);
    printf("b = %d",b);

}
```

Case 2 - Java

```
class Test
{
    public static void main(String args[])
    {
        int a = 10;
        int b = 20;
        int temp;

        System.out.println("a = " +a);
        System.out.println("b = " +b);

        temp = a;
        a = b;
        b = temp;

        System.out.println("a = " +a);
        System.out.println("b = " +b);
    }
}
```



Case 3 - Python

```
a = 10;  
b = 20;  
print("a = ",a);  
print("b = ",b);  
b,a = a,b  
print("a = ",a);  
print("b = ",b);
```

Output-

```
a =10  
b =20  
a =20  
b =10
```

ALL, MLL, HLL

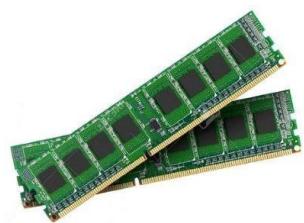
Before you learn any programming language, it is important for one to understand some of the basics about computer and what are the languages that a computer can understand.

Let's have a view on it.

A computer is a collection of hardware components. Let us consider here few hardware components such as:



microprocessor



RAM



Hard disk



GPU

Out of these and many other hardware components, the most important or the heart of the computer is the **Microprocessor or CPU**.

Microprocessor or CPU: A **microprocessor** is an electronic component that is used by a computer to do its work. It is a central processing unit on a single integrated circuit chip containing millions of very small components including transistors, resistors, and diodes that work together. They are created using a technology called as **Semiconductor technology**.

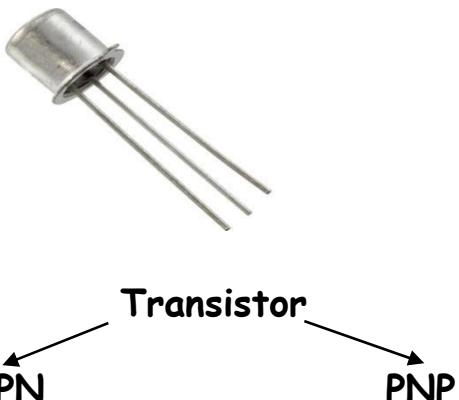
Semiconductor Technology??

Any device which is made up of transistors is referred to as working in Semiconductor Technology.

A **transistor** is a device that regulates current or voltage flow and acts as a switch or gate for electronic signals. The transistors have three terminals emitter, base and collector.

There are two types of transistors:

- 1) NPN transistor.
- 2) PNP transistor.



Transistors can only store voltages. There are two levels of voltages:

Low Voltage referred to as → 0V

High Voltage referred to as → 5V

If we see the same in Software engineer's view, he/she looks the two levels as:

Low Voltage referred to as → 0

High Voltage referred to as → 1

Therefore, in the perspective of a software engineer a Microprocessor or CPU can understand combinations of 0 and 1.

Programming Languages

Language is the main medium for communicating between the computer systems. A program is a collection of instructions that can be executed by a computer to perform a specific task.

There were several programming languages used to communicate with the Computer.

Case-1:

The world's first computer was invented in the year **1940's**.

During that time the task of a programmer was not simple.

For example, if they wanted microprocessor to perform any operation then they had to use combinations of **0's and 1's**.

During this time all the programs where written in the language called as **Machine Level Language**. It is one of the low-level programming languages.



Codes written in 1940's as

To perform addition of two numbers: 0110110

To perform subtraction of two numbers: 1110111

To perform multiplication of two numbers: 1010101

To perform division of two numbers: 0100100

MLL

microprocessor



The machine level code was taken as input and given to the microprocessor as the machine understands the binary value code and it gives the output.

The main **advantage** of using Machine language is that there is no need of a translator to translate the code, as the Computer directly can understand.

The **disadvantage** was, it was difficult for a programmer to write the code or remember the code in this type of language.

Case-2:

The problem with Machine level code approach was decided to be changed in the year 1950's.

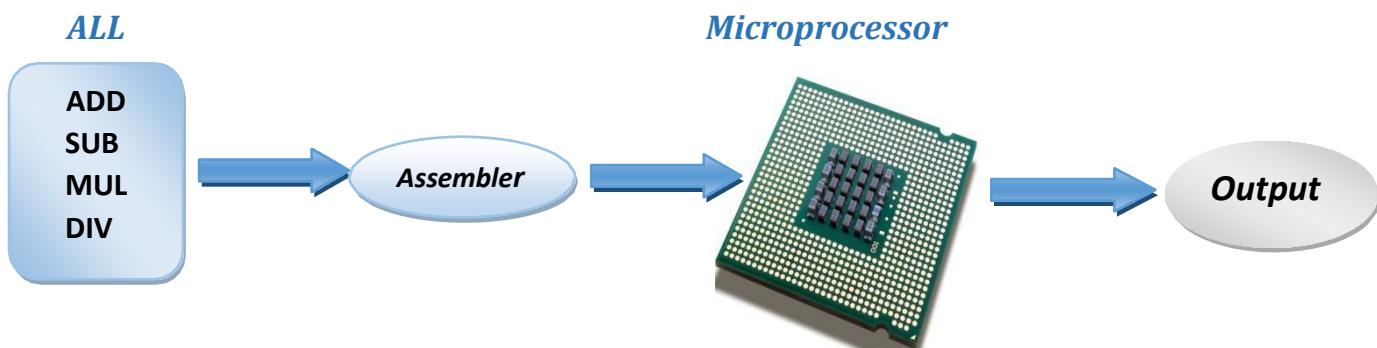
They thought that instead of writing a long sequence of 0's and 1's a **single instruction** can be given.

For example we use,

Codes in 1950's written as

To perform addition of two numbers:	ADD
To perform subtraction of two numbers:	SUB
To perform multiplication of two numbers:	MUL
To perform division of two numbers:	DIV

This approach of writing code is what called as **Assembly Level Language**. Instead of using numbers like in Machine languages here we use words or names in English forms.



An Assembler is software which takes Assembly Level Language (ALL) programs as input and converts it into Machine Level Language (MLL) program.

Case-3:

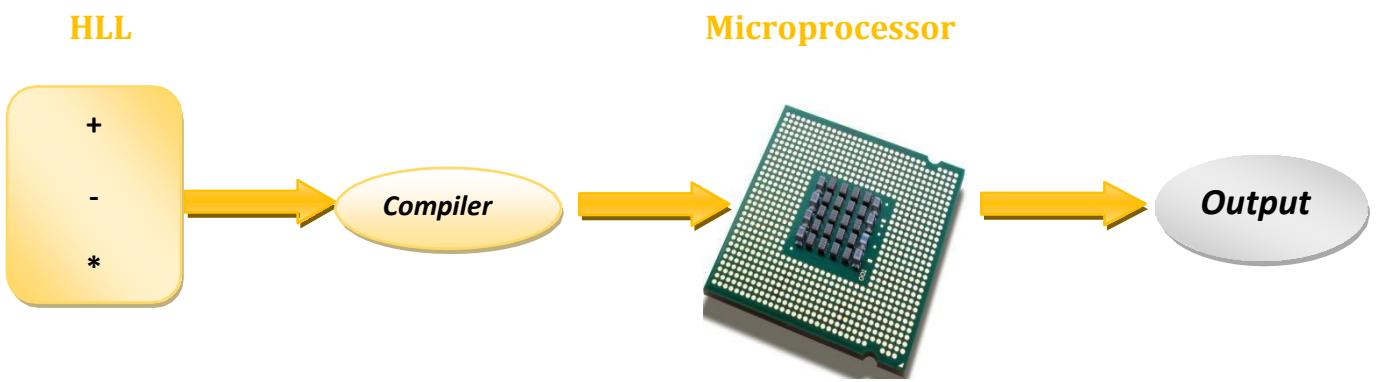
People always want the things to be simple and easier so, in 1960's they came up with next type of language called **High Level Programming Language**.

High Level Languages are written in a form that is close to our human language, enabling the programmer to just focus on the problem being solved.

For example we use,

Codes in 1960's written as

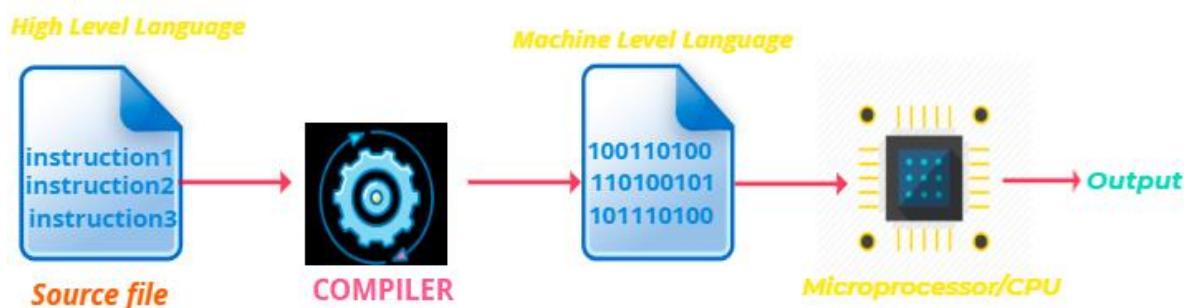
To perform addition of two numbers:	+
To perform subtraction of two numbers:	-
To perform multiplication of two numbers:	*
To perform division of two numbers:	/



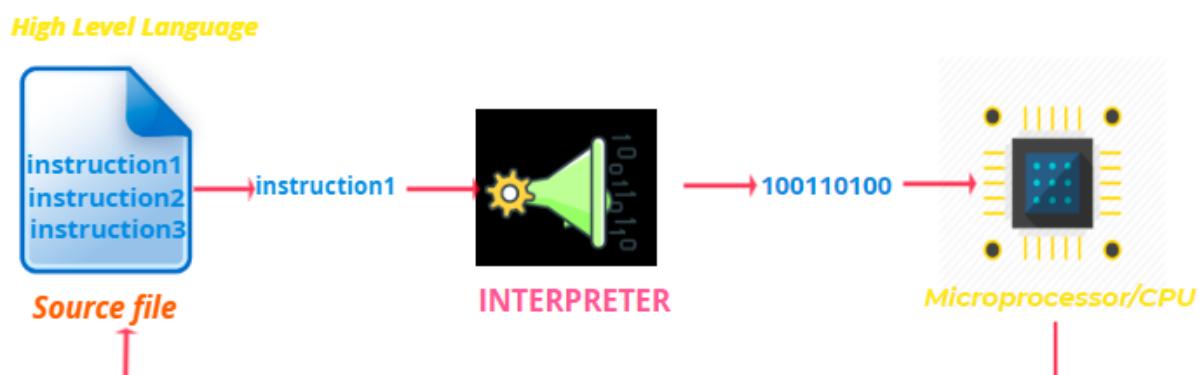
A compiler is software which takes High Level Language (HLL) programs as input and converts it into Machine Level Language (MLL) program.

Compiler vs Interpreter

Compilation



Interpretation



Interpreters and compilers are very similar in structure. The main difference is that compilation is the process of converting a HLL program into MLL using software called as compiler. In compilation the entire HLL program is converted to MLL in one shot by the compiler and hence all the MLL instructions are readily available for CPU.

Interpretation is the process of converting HLL program into MLL using software called as interpreter. In interpretation, at any given point of time only a single line of HLL program is converted to MLL which is then executed by CPU. Post execution the next line is converted and executed. This process repeat itself till all lines in the program are converted and executed. Because of this, CPU does not have all lines of program readily available for execution.

An interpreter will typically generate an efficient Machine Level representation and immediately evaluate it.

Difference between Compiler and Interpreter

No	Compiler	Interpreter
1	Compiler Takes Entire program as input	Interpreter Takes Single instruction as input.
2	Intermediate Object Code is Generated	No Intermediate Object Code is Generated
3	Conditional Control Statements are Executes faster	Conditional Control Statements are Executes slower
4	Memory Requirement : More (Since Object Code is Generated)	Memory Requirement is Less
5	Program need not be compiled every time	Every time higher level program is converted into lower level program
6	Errors are displayed after entire program is checked	Errors are displayed for every instruction interpreted (if any)

Python Fundamentals

day 2

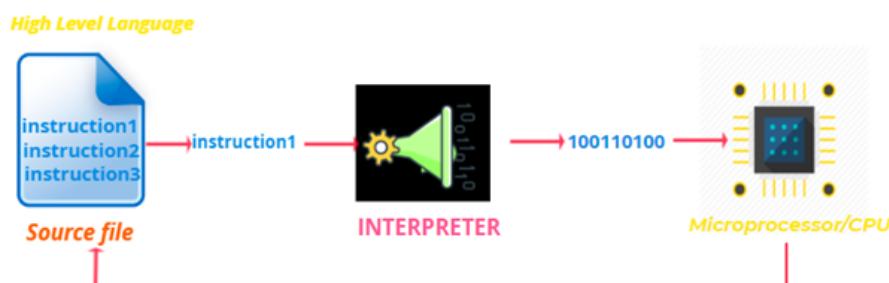
Today's Agenda

- Programming in Python
- Interactive mode & Script mode
- Difference b/w interactive & script mode
- Memory allocation
- Object Oriented Programming
- Principles of OOP



Programming in Python

After knowing what is Python and different features of it and the history behind. Let us now get to know if python is an interpreted programming language or if it is compiled programming language



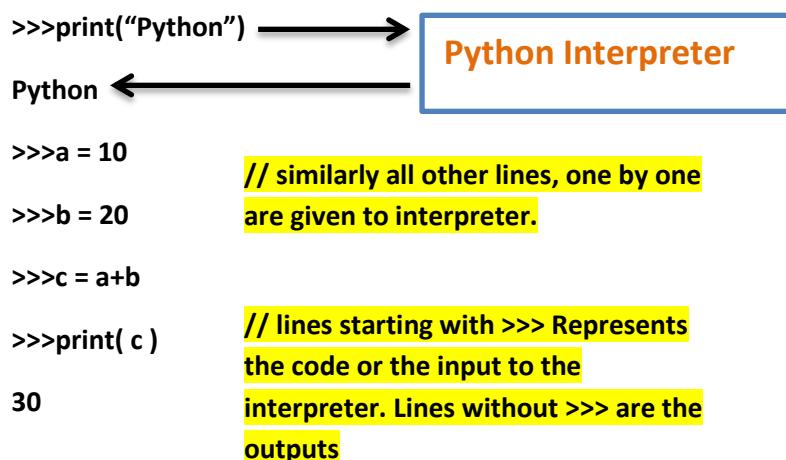
This is how a python file executes. Each instruction is given to the interpreter which converts it into machine level and feeds to CPU to get the output. And again to get the next instruction we have to go back to the source file and continue the process. So no doubt python is an interpreted language.

Interactive mode & Script mode

Let us now look into different modes of writing a code in python:

Interactive mode

The interactive mode involves running your codes directly on the Python shell which can be accessed from the command line/terminal of the operating system. Single line fed to the interpreter is executed at that moment of time, and waits for next line to be entered. Each line is given to the interpreter which internally converts HLL to MLL and the gives the respective output.



And whenever a variable with value is fed to the interpreter a container gets created internally with the value in it. As shown below:

c	30
b	20
a	10

And to exit this interactive mode you should give `quit()` as input.

Pros and cons of Interactive Mode

Pros:

- It is great for single line or smaller codes.
- Interactive mode is a good way to play around and try variations on syntax.

Cons:

- Once you come out of interactive mode, you cannot revisit the previous code.
- It is harder to edit longer programs or even existing program.
- Codes cannot be saved in interactive mode.



Script Mode

To overcome the drawbacks of interactive mode we have the second mode of writing a code called as Script mode. Let us know different platforms of typing a python code in script mode.

- Notepad //basic text editor
- Microsoft Word //basic text editor
- Different IDE (IDLE, Spider, PyCharm, Jupyter Notebook)

Here let us get to know how to run a python script in command line.

Script mode in command line

We know that if we type python and press enter we are into interactive mode like shown below:

```
C:\Users\rooman>python
Python 3.7.4 (default, Aug  9 2019, 18:34:13) [MSC v.1915 64 bit (AMD64)] :: Anaconda, Inc. on win32
Warning:
This Python interpreter is in a conda environment, but the environment has
not been activated. Libraries may fail to load. To activate this environment
please see https://conda.io/activation

Type "help", "copyright", "credits" or "license" for more information.
>>> //entered the interactive mode
```

Let us type the same addition code in python script and see how it works.

```
print("Python")
a = 10
b = 20
c = a+b
print(c)
```

Let us see how to run a python script in command line.

Syntax:

```
C:\Users\rooman>python <python script name>.py
```

Note: Before running this command, make sure you are in the same directory as of the python script you want to run.

Output:

```
C:\Users\rooman>cd OneDrive
```

```
C:\Users\rooman\OneDrive>cd Desktop
```

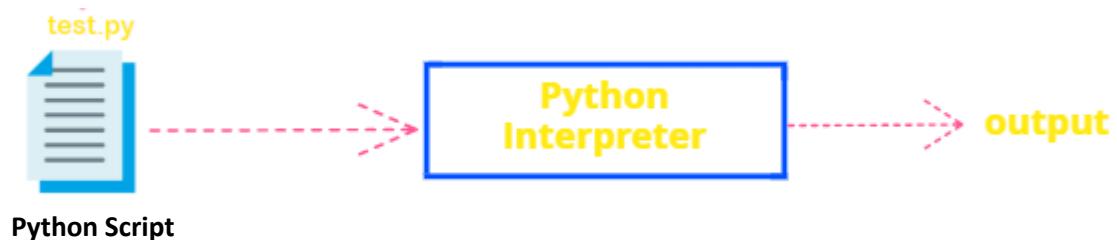
```
C:\Users\rooman\OneDrive\Desktop>cd python
```

```
C:\Users\rooman\OneDrive\Desktop\python>python test.py
Python
30
```

In the script mode, you have to create a file, give it a name with a .py the extension then run your code. The file which contains python code and has to be executed is called as **python script**. The

script mode is recommended when you need to create large applications.

As we have the python script in HLL, conversion must happen as below:



Script mode + Interactive mode

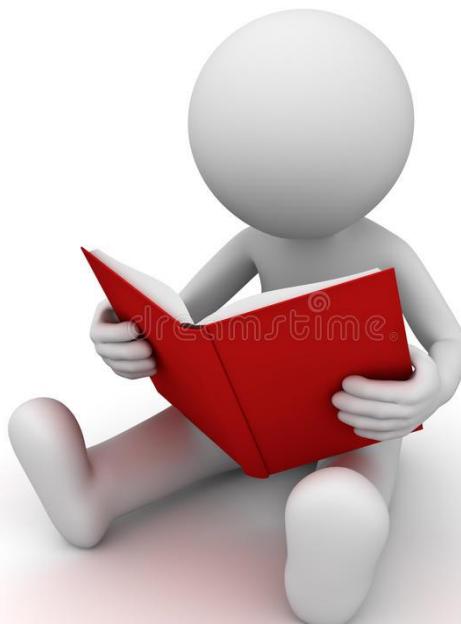
Let us see how to enter interactive mode after running the python script

Syntax:

```
C:\Users\rooman\OneDrive\Desktop\python>python -i <python script>.py
```

Command line execution:

```
C:\Users\rooman\OneDrive\Desktop\python>python -i test.py
Python
30
>>> d = a*b
>>> print(d)
200
```

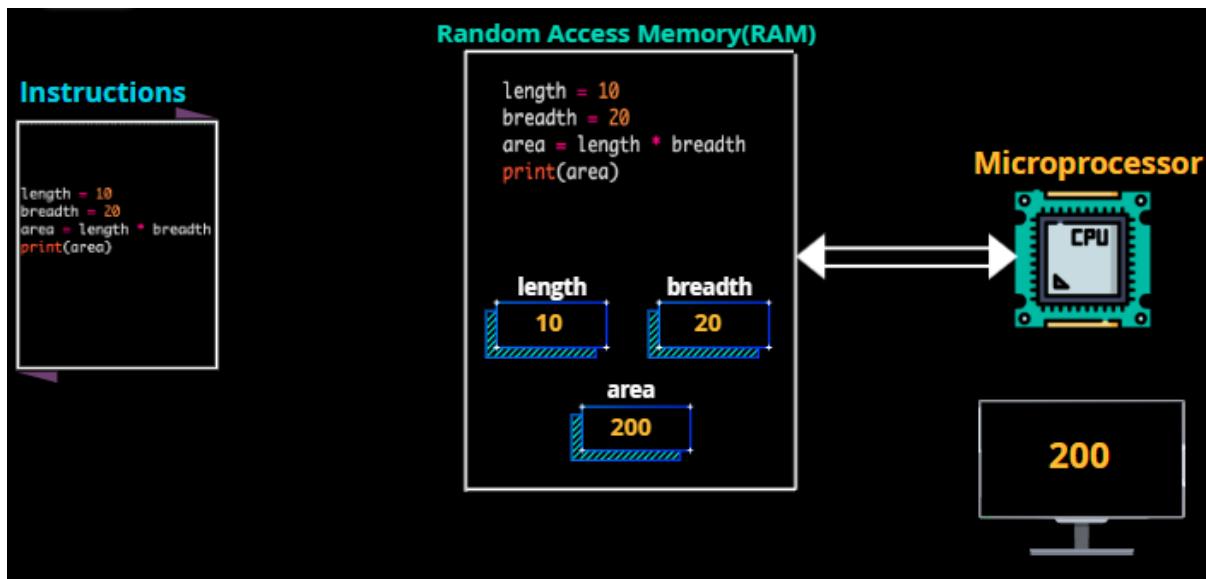


Difference b/w interactive & script mode in python

Interactive mode	Script Mode
A way of using the python interpreter by typing commands & expressions at any prompt.	A way of using the python interpreter to read & execute statements in a script
Can't save and edit the code.	Can save and edit the code
If we want to experiment with the code, we can use interactive mode.	If we are clear about the code, use script mode.
We cannot save the statements for further use and we have to retype.	We can save the statements for further use and no need to retype all the statements to return them.

Memory allocation

On one end we have the instructions and on the other end microprocessor. To give these instructions as the input to microprocessor we need to store it so that the microprocessor has direct access to the instructions. To store these instructions we have something called as **RAM (Random Access Memory)**. RAM is a temporary memory directly connected to the microprocessor, and the main purpose of RAM is to store the instructions before it is given to microprocessor. And then line by line instructions gets converted to MLL and then given to microprocessor.



RAM is the most important memory in the computer where all execution of the program happens on the RAM. In fact if the instructions are on the RAM then only execution can happen if not execution cannot be performed.

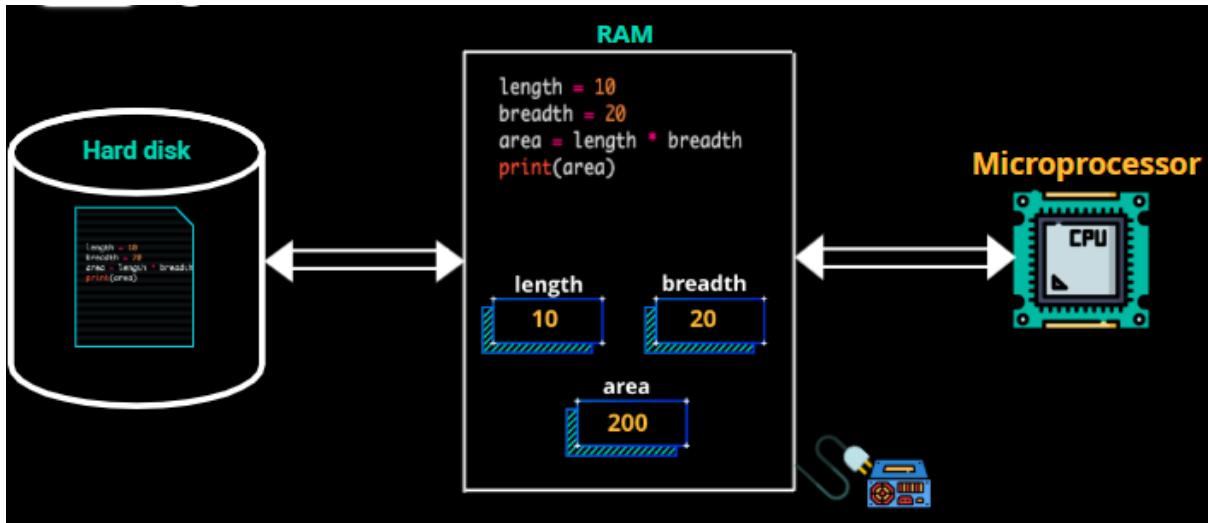
RAM is a semiconductor technology device just like the microprocessor, which means it is made of transistors. So therefore **RAM always needs constant supply of electricity for it to work**. Once the supply is gone all the instructions and operations happening inside the RAM vanishes. That is the reason **RAM is said to be volatile in nature**.



Hard Disk



RAM is the main memory, as it is volatile in nature we have secondary memory which is called as **Hard disk** which is a magnetic device. To use the instructions again and again you need to make a copy of those instructions and store it in hard disk. Those instructions while are stored in hard disk are called as **file**.



The process of taking a copy from RAM and storing in hard disk is called as **saving**. And **loading** is a task where we are taking file and transferring it to RAM.

All computers have these two memory devices namely hard disk and RAM. Where hard disk is used for permanent storage and RAM is used for temporary storage.

Note: Microprocessor is connected only to RAM. And RAM is connected to the hard disk.

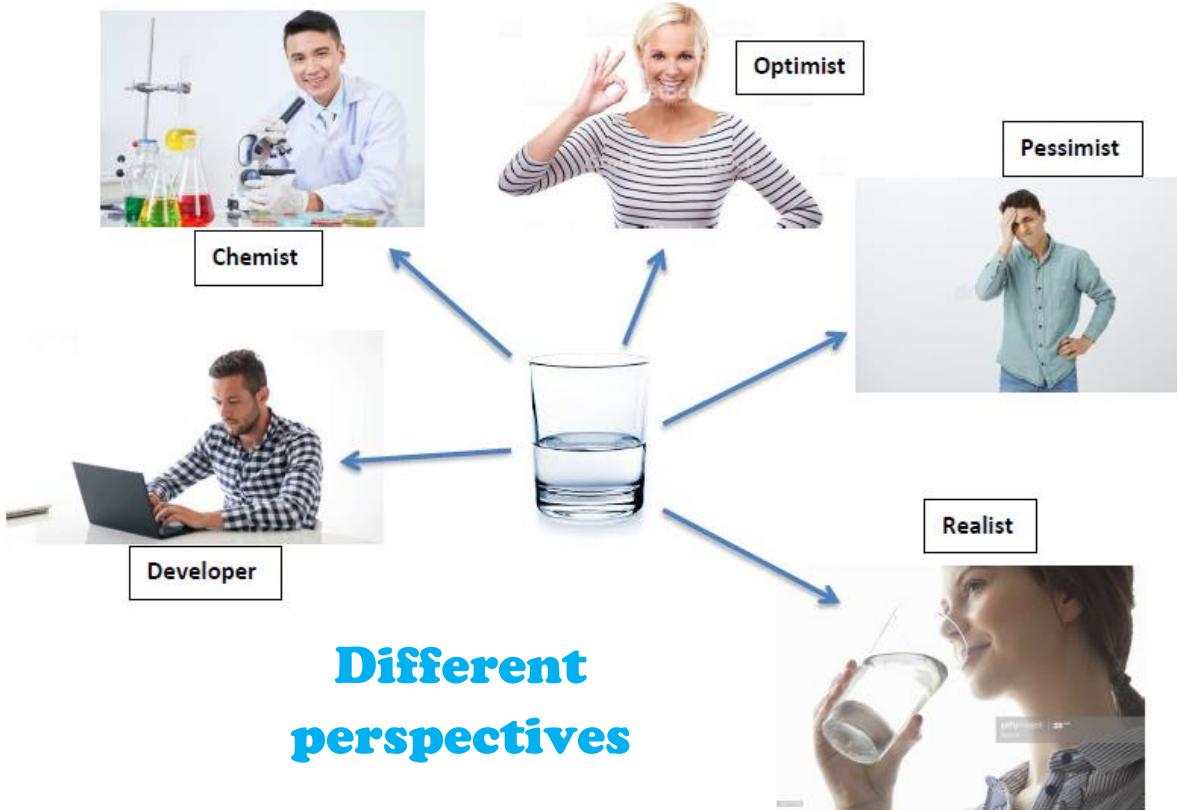
Object Oriented Programming

We have two different types of programming languages, which are structural programming language and the second is object oriented programming (OOP) language. Here we will be focusing on OOP language.

Before going ahead with the **object orientation**, let us get to know what is orientation?

Orientation → Perspective → Way of looking at something

Let us take an example of glass of water and try to understand in better way:



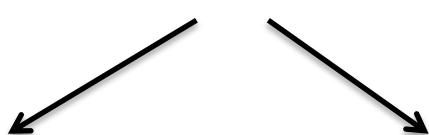
A chemist will see the glass of water as liquid + gas, an optimistic person will look at it as half full glass, a pessimistic person will see it as half empty glass, a realist will see it as glass of water, but a developer will always view it as an object.

Object Orientation is the way of looking at this world as a collection of objects. In this world no object is completely useless. All objects are in constant interaction with each other. No object exists in isolation.

Principles of OOP

- View everything around you as objects.
- Every object belongs to a type. Where type does not exist but the objects of the type exist in reality.
- Every object has two parts:
 1. State of an object /properties
 2. Behaviour of an object/actions
- To handle the properties of the states of an object we have a concept called as **data types**
- Similarly to handle behaviour of an object we have a concept called as **functions**

Let's see the same with example of bike



State/properties:

- Brand
- Price
- Cylinder capacity (cc)
- Colour

Behaviours:

- start()
- change_gear()
- accelerate()
- brake()

Python Fundamentals

day 3

Today's Agenda

- Data types
- Memory Mapping
- Coding - data types



Data types

Before going ahead with what is data type and why do we need it? We should first know **how a data is stored in a system**.

In every electronic system we have **RAM** which **stores the data temporarily**. All the data that we enter is in high level but it is always stored in low level inside the system so that the system/machine can understand.

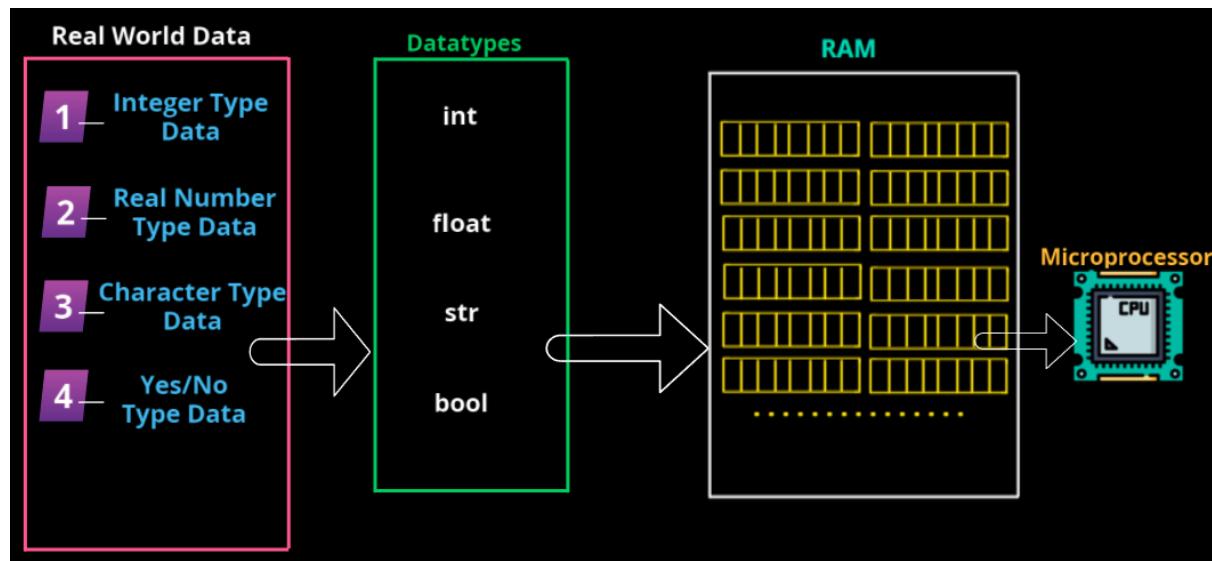
So **RAM** consists of **several bytes**, each byte consists of 8 bits, each bit has two **transistors** which can store **high and low value** (1's and 0's).



Now we know that RAM can only take inputs in 0's and 1's. Apparently the data in real world is not combination of 0's and 1's. We have basic 4 types of real world data which are **integers**, **real numbers**, **characters**, **yes/no types of data**.

The basic data types in python which represents integers, real world, characters, yes/no type data are **int**, **float**, **str**, **bool** respectively.

These data types help to convert high level form of data to combination of 0's and 1's that is binary format.



Examples:

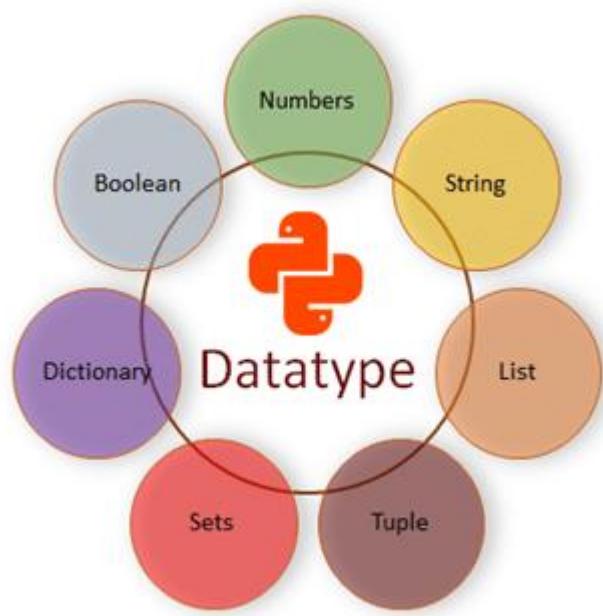
Integers - Age of a person, Number of people, House number, Population of a country, number of galaxies or planets all represent whole numbers. Whole numbers belong to integer type data.

Real world data - Height & weight of a person, GDP of country, your CGPA/SGPA, Literacy rate are all examples of real world data where it may or may not be a whole number. These real world data belong to float data type.

Character - We are surrounded by several things while have names, there isn't a single thing in the world that don't have a name for it. And for communication words play major role. All these are characters which belong to string data type.

Yes/no - We ask several yes or no, true or false questions to one another like married or not? Graduated or not? Employed or not? Literate or not? All these belong to boolean data type.

There are several other data types in python which we shall explore one by one in upcoming sessions.



Memory Mapping

All the operations carried out by computer or any electronic device happens on the RAM. In simple words we can say that **RAM is shared by different applications**. So when we are trying to execute the python code, it won't occupy all the space in RAM. It'll use only a certain region on the RAM. Within this region there are few divisions or segmentations are present. This region contains something called as **stack and private heap**.

In this course all the programs will be explained with respect to these regions.

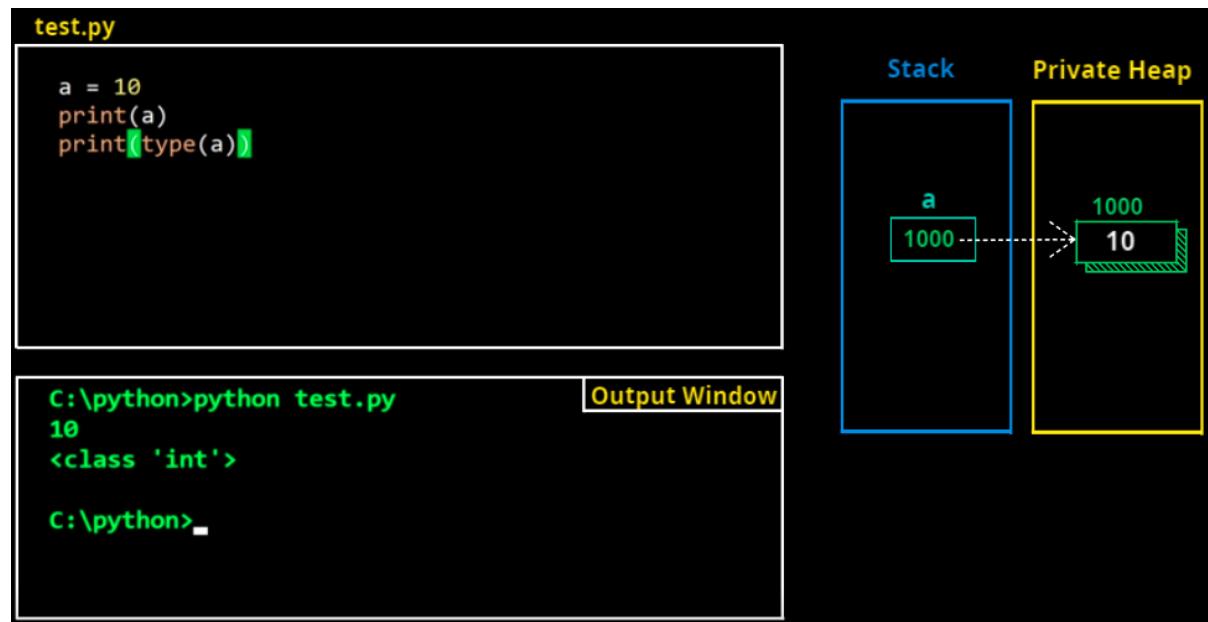
Objects are allocated on private heap. References are allocated on stack. And address of the object created on private heap will be present inside the reference, which means reference is pointing to the object.

Note: As python is dynamically typed programming language, we need not declare the type of object. Based on the value given the language itself decides the type of an object. And in programming languages class means type.

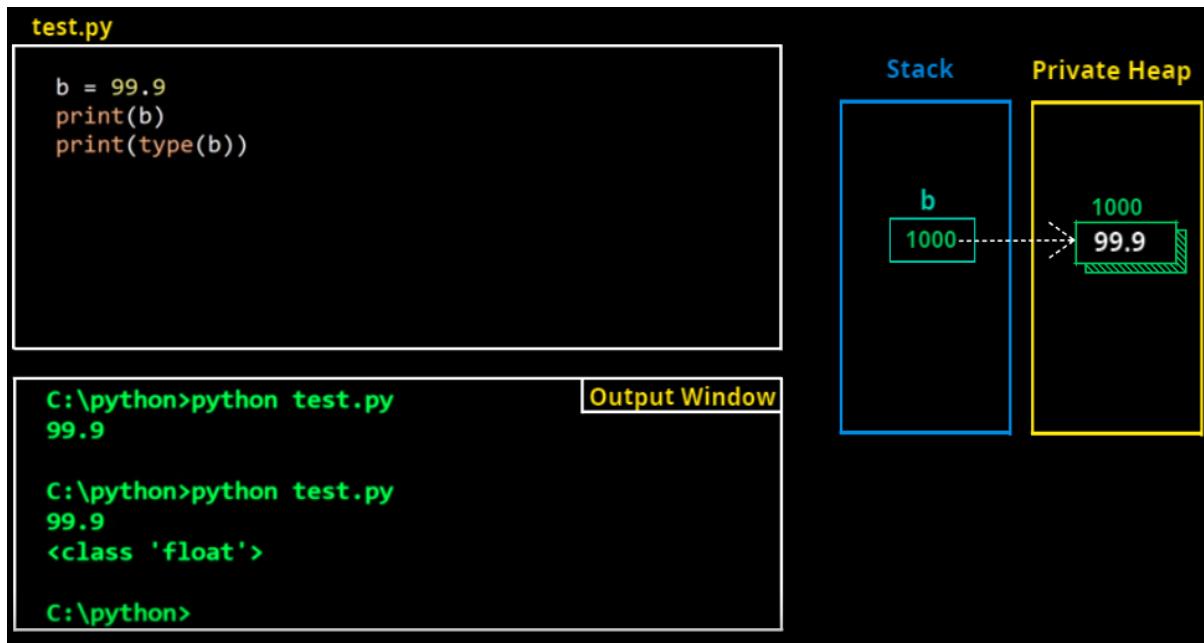


Coding - data types

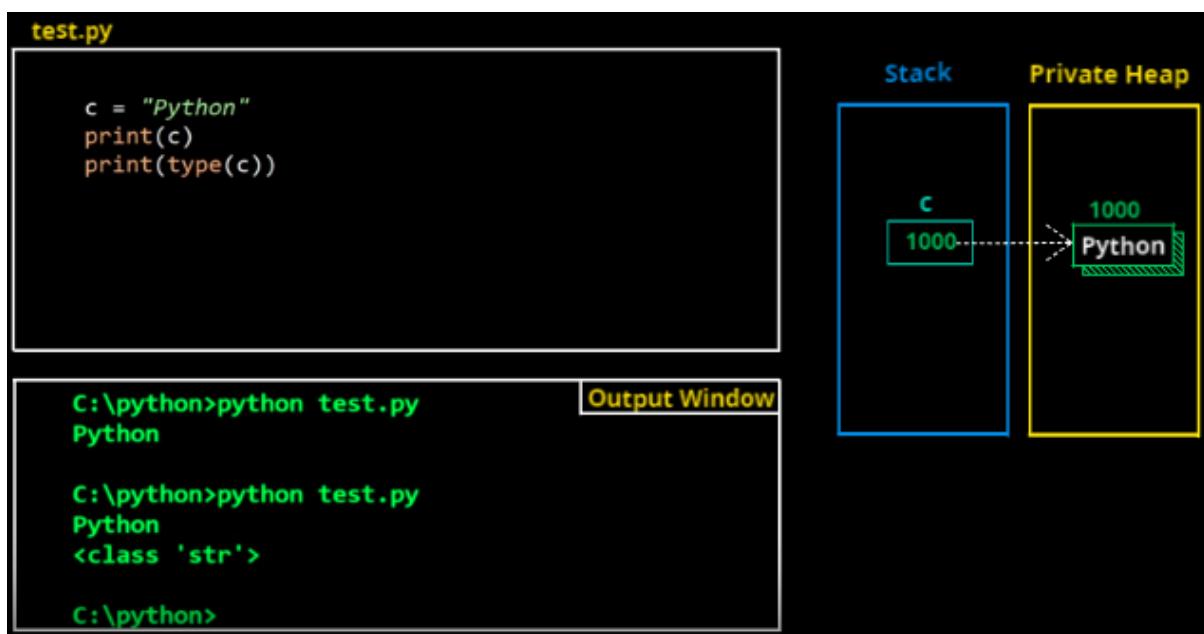
1) Integer data type - int



2) Real world data type - float



3) Character data type - str

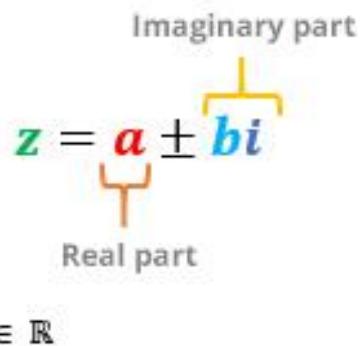


In python language we don't have separate data type for character and string. Here string data type is inclusive of single character type data, collection of characters, and also multi line strings. Which in depth we shall study while learning about strings.

4) Complex numbers -

We have seen the real numbers and integers but there are also imaginary numbers in this world. These imaginary numbers belong to complex numbers.

If you want to know where these imaginary numbers are used in real world then here are few examples, in mathematics I'm sure everyone is aware of quadratic equations and certainly in schooling days we have come across using complex numbers there. And we also use it in electricity, especially in alternating current (AC). Where? How? Why? If these are questions in your mind then you can personally read about it.



$$a, b \in \mathbb{R}$$

The screenshot shows a Python environment with three windows:

- test.py**: A code editor window containing:

```
d = 3+6j
print(d)
print(type(d))
```
- Output Window**: A terminal window showing the execution of the script:

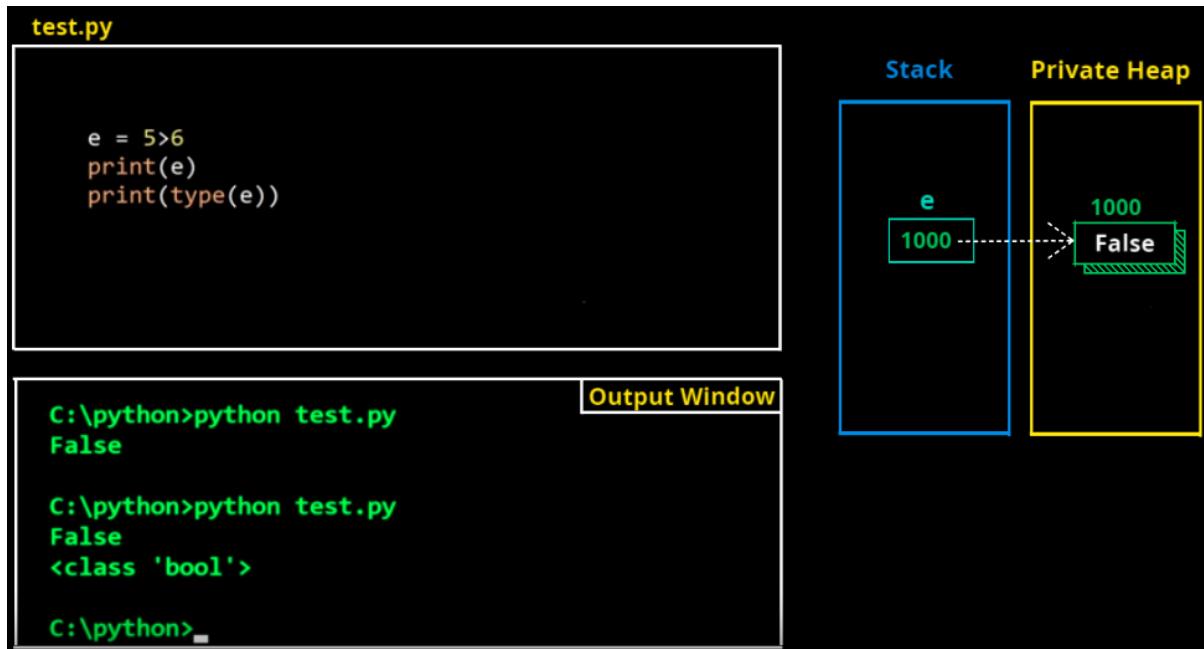
```
C:\python>python test.py
(3+6j)

C:\python>python test.py
(3+6j)
<class 'complex'>

C:\python>
```
- A diagram illustrating memory allocation:
 - Stack**: A blue-bordered box containing a variable **d** with the value **1000**.
 - Private Heap**: A yellow-bordered box containing an object with the value **1000** and a reference to another object with the value **3+6j**.
 - An arrow points from the **d** variable in the Stack to the **3+6j** object in the Private Heap.

5) Boolean - bool

Boolean type of data will have two states true/false. Based on the condition that we want to check, the return value of expression changes. In python Boolean type of data belongs to bool.



Python Fundamentals

day 4

Today's Agenda

- List
- Tuple
- Set
- Dictionary



Lists

A list object is an ordered collection of one or more data items, not necessarily of the same type, put in square brackets.

```
test.py
```

```
lst = [23,56,78,45,13]
print(lst)
print(type(lst))
```

```
C:\python>python test.py          Output Window
[23, 56, 78, 45, 13]

C:\python>python test.py
[23, 56, 78, 45, 13]
<class 'list'>

C:\python>
```

- We can access single elements from list by using positive indices or negative.

```
test.py
```

```
lst = [10,99.9,"Python",True]
print(lst)
print(type(lst))

print(lst[0])
print(lst[-4])
```

lst → [0 1 2 3
-4 -3 -2 -1]

C:\python>python test.py **Output Window**
[10, 99.9, 'Python', True]
<class 'list'>
10
10

C:\python>

- Lists are mutable**, which means we can append elements and remove whenever needed. By using `append()` and `remove()` respectively.

```
test.py
```

```
lst = [10,99.9,"Python",True]
print(lst)
print(type(lst))
lst.append(200)
print(lst)

lst.remove(200)
print(lst)
```

lst → [0 1 2 3
-4 -3 -2 -1]

C:\python>python test.py **Output Window**
[10, 99.9, 'Python', True]
<class 'list'>
[10, 99.9, 'Python', True, 200]
[10, 99.9, 'Python', True]

C:\python>

Tuple

A Tuple object is an ordered collection of one or more data items, not necessarily of the same type, put in parentheses.

- Like lists in tuple also we can access single elements with both positive and negative indices.

```
test.py
t = (10,99.9,"Python",True)
print(t)
print(type(t))
print(t[0])
print(t[-4])
```

1000

0	1	2	3
10	99.9	Python	True
-4	-3	-2	-1

t →

Output Window

```
In [5]: runfile('C:/python/test.py', wdir='C:/python')
(10, 99.9, 'Python', True)
<class 'tuple'>
10
10

In [6]:
```

- Tuple are immutable**, which means we cannot append or remove any elements from a tuple.

```
test.py
t = (10,99.9,"Python",True)
print(t)
#print(type(t))
#print(t[0])
#print(t[-4])
t.append(200)
print(t)
```

1000

0	1	2	3	4
10	99.9	Python	True	200
-5	-4	-3	-2	-1

t →

Output Window

```
In [3]: runfile('C:/python/test.py', wdir='C:/python')
(10, 99.9, 'Python', True)
Traceback (most recent call last):
  File "C:\python\test.py", line 6, in <module>
    t.append(200)
AttributeError: 'tuple' object has no attribute 'append'
```

Set

- In set we cannot access single elements because internally set does not store the data in the order we provide it. That is the reason it is called unordered collection of data. And as it is unordered data, certainly it doesn't have indices.

The diagram shows a set `s` containing the elements 40, 10, 50, 20, and 30. An arrow labeled `s` points to the first element, which is highlighted with a green border. The index 0 is shown above the first element, and the value 40 is shown above the index 0. The other four elements are also highlighted with green borders.

```
test.py
s = {10,20,30,40,50}
print(s)
print(type(s))
print(s[0])
```

In [4]: runfile('C:/python/test.py', wdir='C:/python')
{40, 10, 50, 20, 30}
<class 'set'>
Traceback (most recent call last):
 File "C:/python/test.py", line 4, in <module>
 print(s[0])
TypeError: 'set' object is not subscriptable

Output Window

- Set is mutable**, which means we can add or remove the elements using `add()` and `remove()` respectively. But remember that the order in which it adds an element is not as we enter it.

The diagram shows a set `s` containing the elements 40, 10, 50, 20, 30, and 200. An arrow labeled `s` points to the first element, which is highlighted with a green border. The index 0 is shown above the first element, and the value 40 is shown above the index 0. The other five elements are also highlighted with green borders.

```
test.py
s = {10,20,30,40,50}
print(s)
print(type(s))
#print(s[0])
s.add(200)
print(s)
s.remove(200)
print(s)
s.add(50)
print(s)
```

In [8]: runfile('C:/python/test.py', wdir='C:/python')
{40, 10, 50, 20, 30}
<class 'set'>
{40, 200, 10, 50, 20, 30}
{40, 10, 50, 20, 30}
{40, 10, 50, 20, 30}

In [9]:

Output Window

- **No duplication**, which means the data inside set are unique. Although duplicated entries will not give an error, the entries will not get stored and be seen on screen.

test.py

```
s = {10, 20, 30, 40, 50}
print(s)
print(type(s))
#print(s[0])
s.add(200)
print(s)
s.remove(200)
print(s)
```

1000

`s` → [40 | 10 | 50 | 20 | 30]

Output Window

```
In [7]: runfile('C:/python/test.py', wdir='C:/python')
{40, 10, 50, 20, 30}
<class 'set'>
{40, 200, 10, 50, 20, 30}
{40, 10, 50, 20, 30}

In [8]:
```

Dictionary

A dictionary is a collection which is unordered, changeable and indexed. In Python dictionaries are written with curly brackets, and they have keys and values. Each key will be mapped to a value.

test.py

```
d = {"Dennis Ritchie": "C",
"James Gosling": "Java",
"Guido Van Rossum": "Python"}
print(d)
print(type(d))
```

key	value
Dennis Ritchie	C
James Gosling	Java
Guido Van Rossum	Python

`d` → [key | value]

Output Window

```
In [3]: runfile('C:/python/test.py', wdir='C:/python')
{'Dennis Ritchie': 'C', 'James Gosling': 'Java', 'Guido Van Rossum': 'Python'}
<class 'dict'>

In [4]:
```

Dictionaries are mutable - which means we can add and remove keys. A key can be removed by using `pop()`. Many other operations can be done as well, which we shall learn in future sessions.

```
test.py
d = {"Dennis Ritchie": "C",
      "James Gosling": "Java",
      "Guido Van Rossum": "Python"}
print(d)
#print(type(d))
d["Brendan Eich"] = "Javascript"
print(d)
d.pop("Brendan Eich")
print(d)
```

`d` →

key	value
Dennis Ritchie	C
James Gosling	Java
Guido Van Rossum	Python

Output Window

```
In [5]: runfile('C:/python/test.py', wdir='C:/python')
{'Dennis Ritchie': 'C', 'James Gosling': 'Java', 'Guido Van Rossum': 'Python'}
{'Dennis Ritchie': 'C', 'James Gosling': 'Java', 'Guido Van Rossum': 'Python',
 'Brendan Eich': 'Javascript'}
{'Dennis Ritchie': 'C', 'James Gosling': 'Java', 'Guido Van Rossum': 'Python'}
```

In [6]:

Python Fundamentals

day 5

Today's Agenda

- Typecasting
- Mutable and Immutable datatypes
- Overview of datatypes



Typecasting

The process of converting one type of data to another type in programming language is called as typecasting. In python typecasting is done using the functions already present in python these functions are called as built-in functions.

`int()` `str()` `list()` `set()` `hex()` `chr()`

`float()` `complex()` `tuple()` `dict()` `oct()` `ord()`

From the above mentioned built-in functions let's focus on the basic datatypes for now, rest of them we shall see in later on sessions.

- First conversion:

`Integer` → `Float`
`float()`

Integer value must be converted to a floating point value, if this must happen then we have to use a function which converts integer type data to float type data. Which is none other than float().



```
In [1]: a=99  
In [2]: print(a)  
99  
  
In [3]: print(type(a))  
<class 'int'>  
  
In [4]: b=float(a)  
  
In [5]: print(b)  
99.0  
  
In [6]: print(type(b))  
<class 'float'>
```

- **Second conversion:**

Float → Integer
int()

Let's reverse the order now by converting floating type data to integer type data by using int().

```
In [7]: a=99.9  
In [8]: print(a)  
99.9  
  
In [9]: print(type(a))  
<class 'float'>  
  
In [10]: b=int(a)  
  
In [11]: print(b)  
99  
  
In [12]: print(type(b))  
<class 'int'>
```



Note: We observe that 0.9 is lost during the conversion, because integer type data will not store decimal point values. Therefore we have to be very careful during type casting. In some cases 0.9 is a very huge value whereas in some other cases it is completely fine.

- **Third conversion:**

Float → Complex
complex()



Let us convert a floating number to a complex number which contains a real number and an imaginary number.

```
In [13]: a=99.9
In [14]: print(a)
99.9
In [15]: print(type(a))
<class 'float'>
In [16]: b=complex(a)
In [17]: print(b)
(99.9+0j)
In [18]: print(type(b))
<class 'complex'>
```

We can also give an imaginary number as shown below

```
In [19]: c=complex(a,4)
In [20]: print(c)
(99.9+4j)
```

- **Fourth conversion:**

Integer → String
str()

We have many other conversions which we will perform as and when needed. But let us see the last conversion which is most commonly used, integer type data to string type.

```
In [21]: a=99
```

```
In [22]: print(a)
99
```

```
In [23]: print(type(a))
<class 'int'>
```

```
In [24]: b=str(a)
```

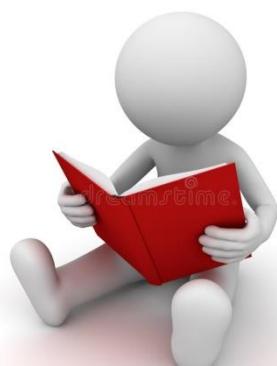
```
In [25]: print(b)
99
```

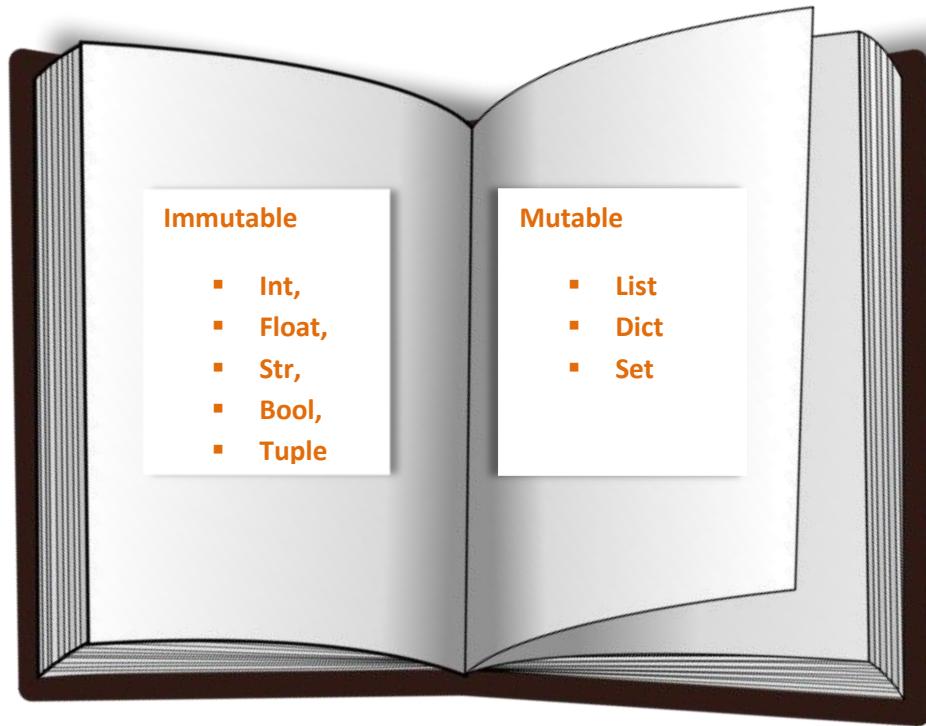
```
In [26]: print(type(b))
<class 'str'>
```

String type data can certainly store numbers as well. We can know the type by printing it.

Mutable & Immutable datatypes

As we already know that mutable means changeable and immutable means non changeable. The datatypes whose values cannot be changed are called immutable datatypes and likewise the datatypes whose values can be changed are called mutable datatypes.

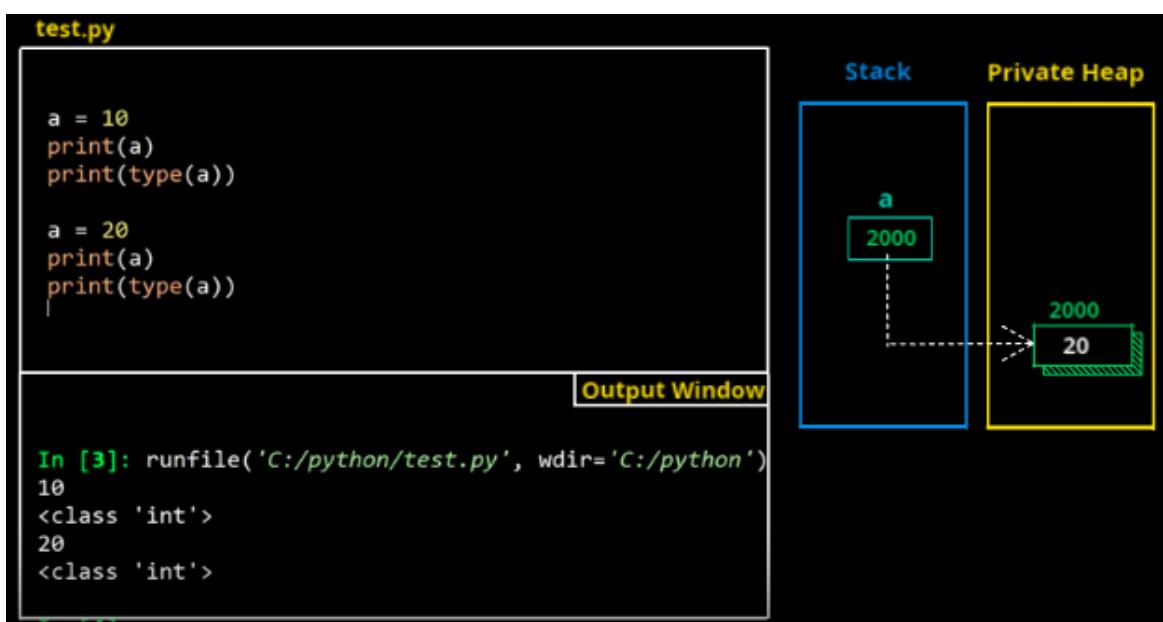




▪ **Immutable datatypes - (Integer)**

Let us try assigning a value to variable `a` and try to change values.

In below diagram we can see that `a` value gets changed but internally two different objects are created with different address. So the variable `a` is now pointing to the new value.



Let us try to verify the above statement by printing the address of both the objects.

The diagram illustrates the memory layout in Python. On the left, a code editor window titled "test.py" shows the following Python script:

```
a = 10
print(a)
print(type(a))
print(id(a))
a = 20
print(a)
print(type(a))
print(id(a))
```

On the right, the Jupyter Notebook interface shows the output of running this script in cell [5]. The output is:

```
In [5]: runfile('C:/python/test.py', wdi=Output Window
10
<class 'int'>
140717117776560
20
<class 'int'>
140717117776880
```

Below the output, a memory diagram is shown. It consists of two vertical columns: "Stack" and "Private Heap". In the Stack column, there is a box labeled "a" with the value "2000" below it. A dashed arrow points from this "a" box to a box in the Private Heap column labeled "2000" with the value "20" below it. This indicates that the variable "a" in the stack contains a reference to the integer object "20" in the private heap.

We can see from the above diagram that both objects have different references. Which implies, when we try to modify the values assigned to a variable, new object gets created instead of modifying the existing value.

It is the same feature of immutability of certain datatypes which makes python memory efficient in the background. Which simply means that, when two variables are assigned the same value, two objects does not get created on private heap instead both the variables will be pointing to the same object. Because by now we know that some datatypes are immutable and their values cannot be changed. This is demonstrated in the below diagram,

```

test.py
a = 10
print(a)
print(type(a))
b = 10
print(b)
print(type(b))
print(id(a))
print(id(b))
print(a is b)

Output Window
10
<class 'int'>
10
<class 'int'>
140717117776560
140717117776560
True

In [6]:

```

Above we have shown with respect to integer type data, whereas the same works for float, bool, string and tuple also because all these belong to immutable type data. And certainly we have also checked for equality of two variables by `is` operator.

- **Mutable datatype - (list)**

Let us create a list having some set of values and try to append new values. If you have gone through previous sessions you know that certainly that is possible in lists. Let's see it either ways

```

test.py
lst = [23,56,78,45,13]
print(lst)
print(type(lst))

lst.append(200)
print(lst)

Output Window
In [4]: runfile('C:/python/test.py', wdir='C:/python')
[23, 56, 78, 45, 13]
<class 'list'>
[23, 56, 78, 45, 13, 200]

In [5]:

```

Let us now try to create another list with same values and see if memory is efficient in this case as well

The screenshot shows a Jupyter Notebook interface. On the left, a code cell named 'test.py' contains the following Python code:

```
lst1 = [23, 56, 78, 45, 13, 200]
print(lst1)
print(type(lst1))

print(lst1 is lst1)
```

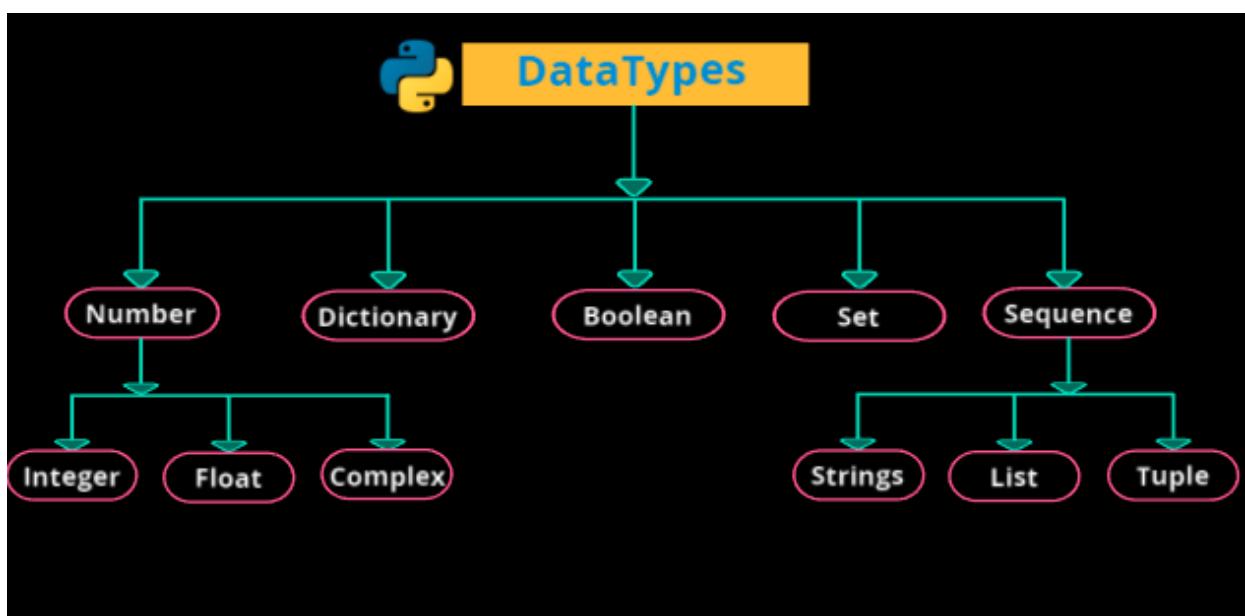
On the right, the 'Output Window' shows the execution results:

```
In [8]: runfile('C:/python/test.py', wdir='C:/python')
[23, 56, 78, 45, 13]
<class 'list'>
[23, 56, 78, 45, 13, 200]
[23, 56, 78, 45, 13, 200]
<class 'list'>
False
```

Below the output window, a diagram illustrates the memory state. It shows two variables, 'lst' at address 1000 and 'lst1' at address 2000, both pointing to the same list object at address 2000. The list object contains the values 23, 56, 78, 45, 13, and 200.

We can see that when new list `lst1` was equated to `lst` it resulted as `false` (Equality basically checks the references). This is because lists are mutable. So if any changes are made to one list the other list might get affected too. That's the reason no two lists type data will be pointing to a same lists object.

Overview of datatypes



Python Fundamentals

day 6

Today's Agenda

- Introduction to Functions
- Why Functions?
- Different types of functions
- User Defined functions in detail



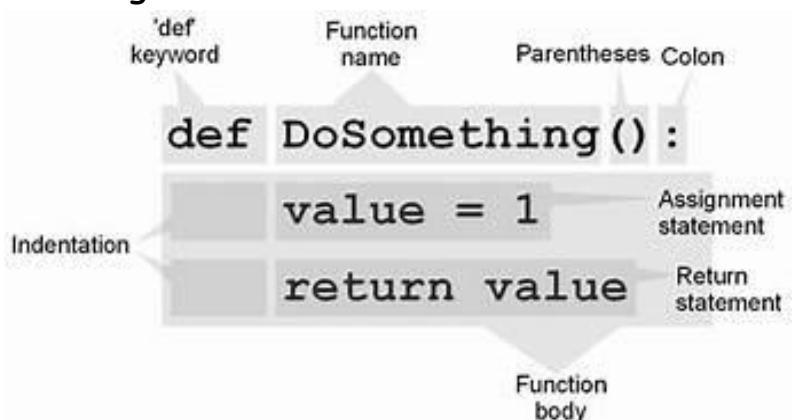
Introduction to Functions

Definition: A Function is a set of statements that takes inputs, performs some specific task and produces output.

How do i declare a function in python?

The four steps to defining a function in Python are the following:

- Use the keyword **def** to declare the function and follow this up with the **function name**.
- Add **parameters** to the function: they should be within the parentheses of the function. End your line with a **colon**.
- Add **statements** that the functions should execute.
- End your function with a **return statement** if the function should output something.



Why Functions?

After getting to know how a python function looks like, let us now understand why functions by considering few examples.

Let us consider two operations of adding two numbers and dividing two numbers as shown below.

```
a = 10  
b = 20  
c = a+b  
print(c)
```

```
d = 100  
e = 10  
f=d/e  
print(f)
```



Above lines of codes will perform addition and division of two numbers, but if in case you want to perform addition of two numbers again, the only option is to copy the same lines of code and use them again as shown below:

```
a = 10  
b = 20  
c = a+b  
print(c)
```

```
d = 100  
e = 10  
f=d/e  
print(f)
```



```
a = 10  
b = 20  
c = a+b  
print(c)
```



But as a programmer this is **not the right approach** which one should follow, the efficient way of doing this is by following **dry approach** which stands for **DO-NOT-REPEAT-YOURSELF.**

So, the solution is including the lines of code inside a block and give it a suitable name. That way you can use the same block of code multiple times by making use of name given to it.

Using the above approach, we can achieve:

Reusability and Modularity.

Reusability refers to resuing the same block of codes multiple times.

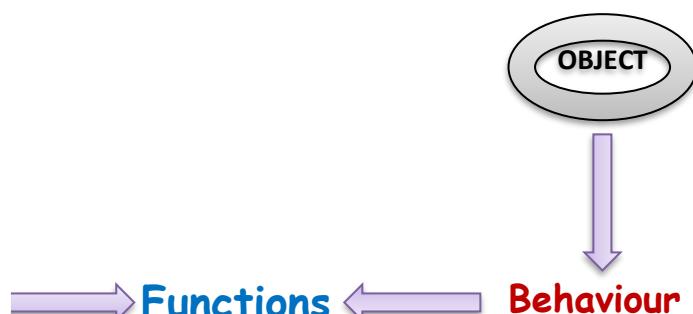
Modularity refers to dividing the entire code into different logical blocks of code.

From object orientation perspective, **the behaviour of an object** is taken care by **functions** in python.

```
a = 10  
b = 20  
c = a+b  
print(c)
```

```
d = 100  
e = 10  
f=d/e  
print(f)
```

- 1. Reusability**
- 2. Modularity**



Types of Functions in python

Python consists of four types of functions:

1. User Defined Functions
2. Built-in-functions.
3. Lambda functions.
4. Recursive Functions.



User Defined Functions: These functions are defined or created by user.

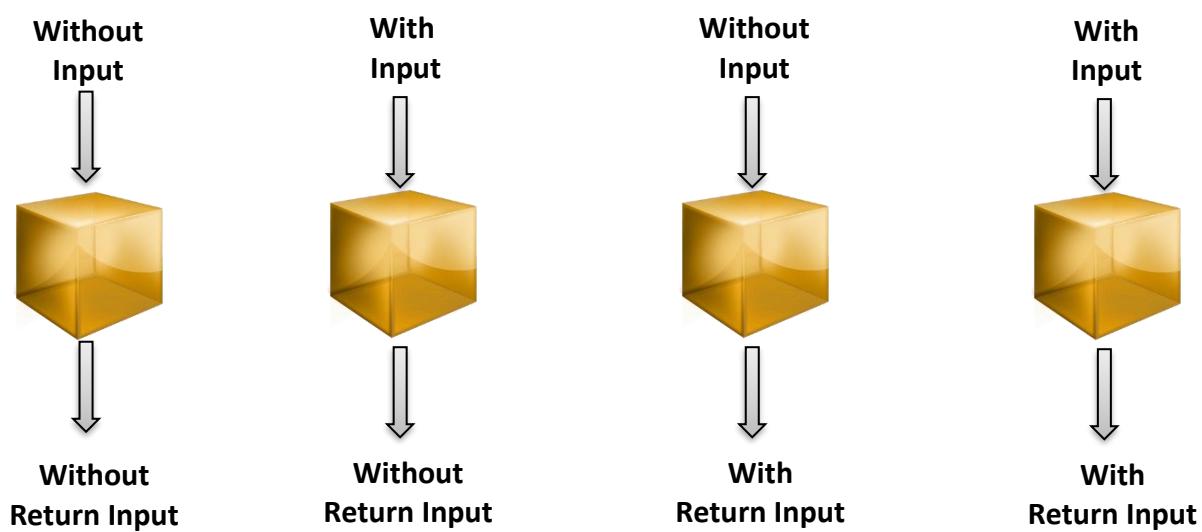
Built-in-functions: These functions are already defined in Python libraries and we can call them directly.

Lambda functions: A lambda function can take any number of arguments, but can only have one expression.

Recursive Functions: A Python function which is defined to call itself during its execution is known as python recursive function.

User defined Functions

A user defined function in python can be declared in four different ways as shown below:



1. Function that takes no input and does not return any output.

Below is the code which consists of a function `mul()` which falls in the first category of user defined functions which is a function that takes no input and returns no output.

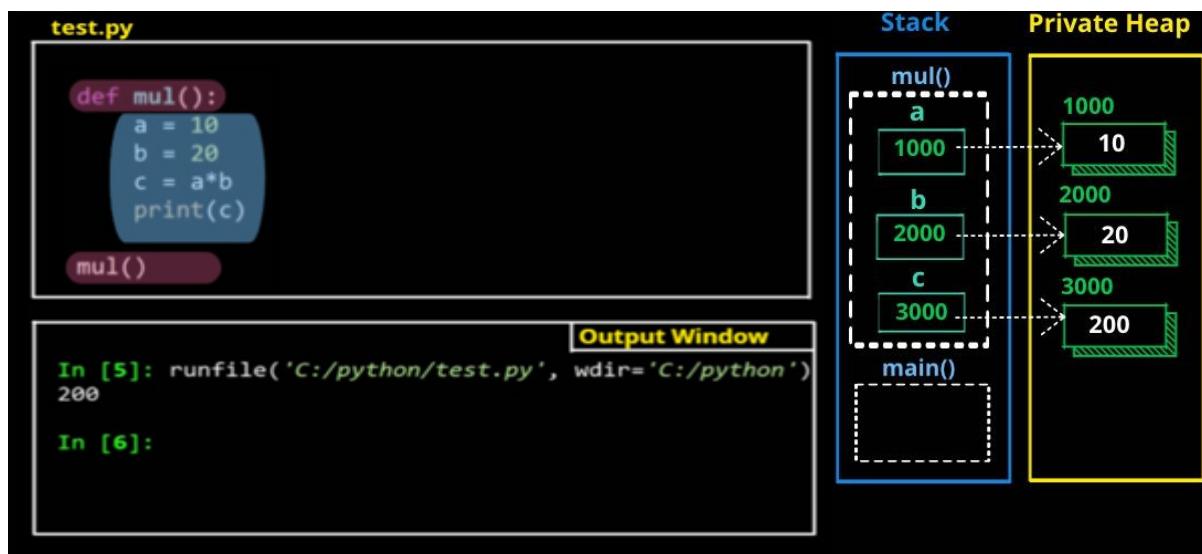
In the code given below there is not only `mul()`, it also has `main()` inside which we are calling `mul()`.

All such statements which are not present inside any functions are automatically placed within the `main()` function and is automatically called in python. The moment a method gets called its stack frame gets created.

The first function that gets called in a python program is always the `main()`. Whenever a main function gets called its stack frame gets created on stack which simply shows `main()` has begun execution.

Inside `main()`, we are calling `mul()` which results in the creation of stack frame of `mul()` on stack.

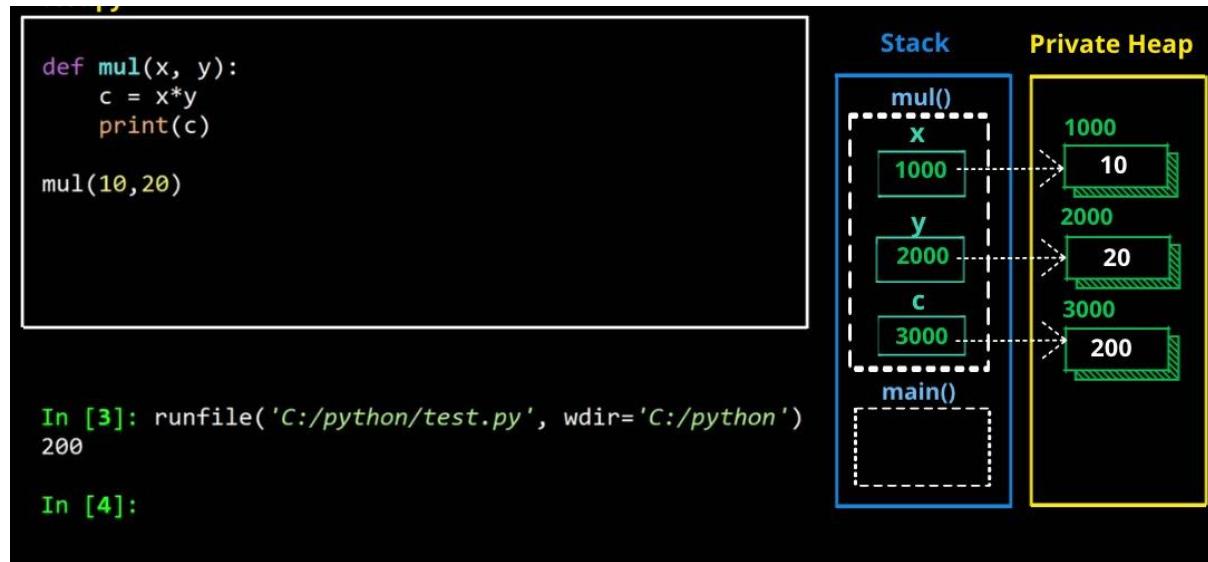
Whatever is present at the top of stack is always the one which is currently executing and hence `mul()` starts executing inside which we are creating three objects `a,b,c`. The references are created inside the stack frame of `mul()` and objects are created on private heap. Once `mul()` finishes execution, its stack frame gets deleted and then the control is given back to `main()`. After the execution of `main()`, its stack frame also gets deallocated memory on stack. If all the stack frames are destroyed, all the references also gets deleted. Now the objects on heap do not have any reference pointing to them, once python encounters there no references pointing to objects, it treats those objects as garbage objects and deletes them. To perform deletion of garbage objects, python provides a software called as garbage collector which deallocates memory for all the garbage objects.



2. Function that takes input and does not return any output.

Below is the code which consists of a function `mul()` which falls in the **second category** of user defined functions which is a function that takes input and returns no output.

While calling a method that requires arguments, **one must pass number of arguments accepted by the methods** as shown below where we are passing 10,20 while calling `mul(10,20)`. 10 and 20 are now stored inside x and y.



3. Function that takes no input but returns output.

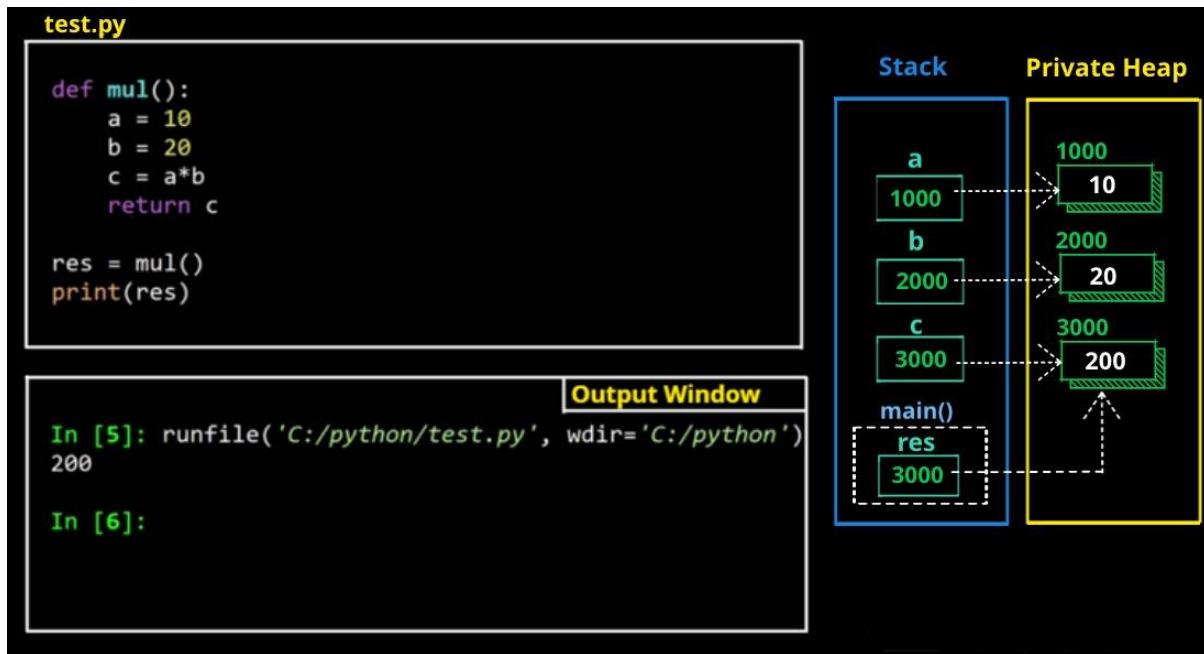
Below is the code which consists of a function `mul()` which falls in the **third category** of user defined functions which is a function that takes no input and returns output.

One must make use of **return statement** inside the function to return a value.

Once return statement gets executed, **control goes to the statement which had called that function**. Now it is the choice of the caller to collect the value returned by function or not. In the below code the returned value is collected inside `res` which is created inside the stack frame of `main()` as shown in figure below.



Note: Python functions can return multiple values.



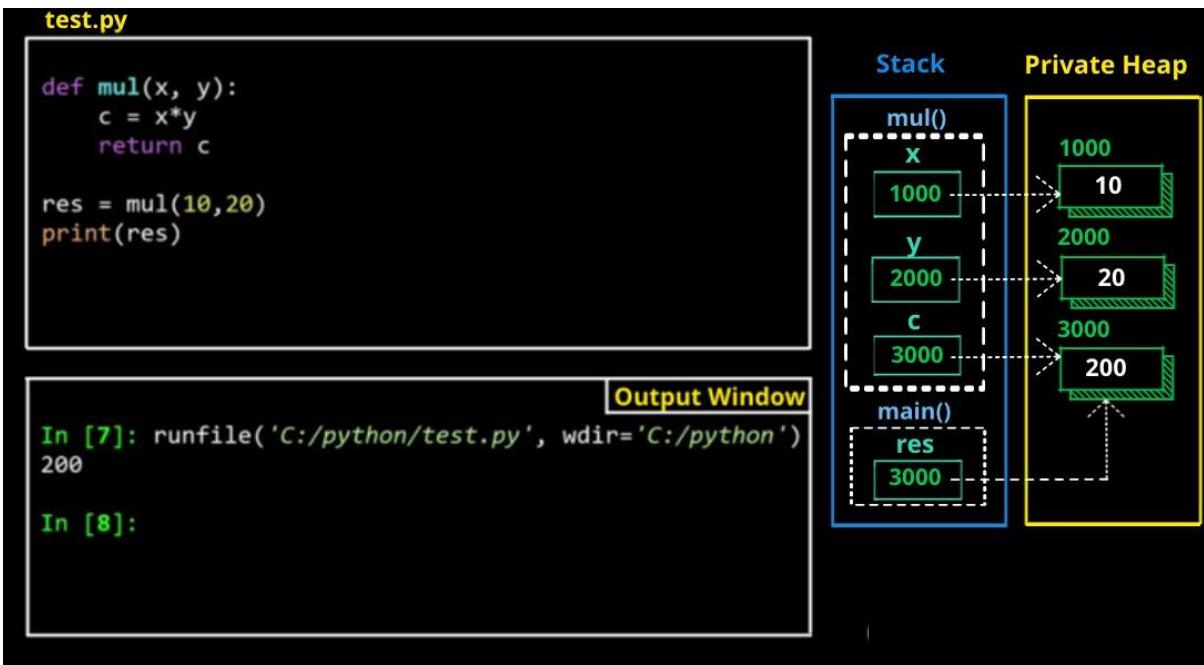
4. Function that takes input and returns output

Below is the code which consists of a function `mul()` which falls in the **fourth category** of user defined functions which is a function that takes two inputs `x,y` and returns output.

Based on situations, we must make use of different functions.

For example, you need your bank account details and to do so you must give your details as input upon which your details will be returned to you.

This way, based on different scenarios, different types of functions should be used to achieve the required task.



Python Fundamentals

day 7

Today's Agenda

- Types of arguments
- Positional arguments
- Default arguments
- Keyword arguments
- Variable length arguments
- Variable keyword length arguments



Types of arguments

The input passed through a function is technically called as argument. Let us know better

```
def mul(x,y):  
    c=x*y  
    print(c)  
  
mul(10,20)
```

↗ → **Parameters**

└→ **Arguments**

Parameters collect the inputs, which is passed to the function when it is called.

We have different types of arguments:

1. Positional arguments
2. Default arguments (Optional arguments)
3. Keyword arguments
4. Variable length arguments (Arbitrary arguments)
5. Variable keyword length arguments (Arbitrary keyword arguments)

Positional Arguments

The screenshot shows a Jupyter Notebook interface. On the left, a code cell named 'test.py' contains the following Python code:

```
def power_of(a, b):
    c = a**b
    print(c)

power_of(2,5)
```

Below the code cell is an 'Output Window' containing the command 'In [2]: runfile('C:/python/test.py', wdir='C:/python')' and the output '32'. To the right of the notebook, there is a diagram titled 'Positional arguments' with three boxes labeled 'a', 'b', and 'c' with their corresponding values: 2, 5, and 32.

In the above example we are performing power operation. Here the arguments are **2,5** and parameters are **a,b** where **a** gets assigned as **2** and **b** gets assigned as **5** based on the positions. To verify this let's try to interchange the positions and check whether the value of **c** changes or remains same.

The screenshot shows a Jupyter Notebook interface. On the left, a code cell named 'test.py' contains the following Python code:

```
def power_of(a, b):
    c = a**b
    print(c)

power_of(2,5)
power_of(5,2)
```

Below the code cell is an 'Output Window' containing the command 'In [2]: runfile('C:/python/test.py', wdir='C:/python')' and the output '32'. Below it, another command 'In [3]: runfile('C:/python/test.py', wdir='C:/python')' has been entered but not yet run. To the right of the notebook, there is a diagram titled 'Positional arguments' with three boxes labeled 'a', 'b', and 'c' with their corresponding values: 5, 2, and 25.

As we can see, **a** gets assigned as **5** and **b** gets assigned as **2**. Which proves the above statement, if passed in the above format, position definitely matters.

If we miss a single argument then error appears, let us see what the error is

test.py

```
def power_of(a, b):
    c = a**b
    print(c)

power_of(2,5)
power_of(5,2)
power_of(2)
```

Positional arguments

Output Window

```
In [4]: runfile('C:/python/test.py', wdir='C:/python')
32
25
Traceback (most recent call last):
  File "C:/python/test.py", line 7, in <module>
    power_of(2)
TypeError: power_of() missing 1 required positional argument: 'b'
```

Error clearly tells us that we are missing an argument **b**. So now let us see how to get away with this.

Default Arguments

test.py

```
def power_of(a, b=0):
    c = a**b
    print(c)

power_of(2)
```

Default arguments

Output Window

```
In [6]: runfile('C:/python/test.py', wdir='C:/python')
1
```

The diagram shows three boxes labeled 'a', 'b', and 'c' with their corresponding values: 'a' has value '2', 'b' has value '0', and 'c' has value '1'. This illustrates how Python handles default parameter values.

In the above example we have only single argument, assuming **b** will take the default value assigned to it as **0**. Giving a value while declaring the parameter is referred to as default value.

Certainly we can also give two arguments

```
def power_of(a,b=0):  
    c=a**b  
    print(c)  
  
power_of(2)  
power_of(2,5)
```



Output:

```
In [30]: runfile('C:/Users/rooman/OneDrive/Desktop/python/  
test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')  
1  
32
```

Let us see another example

```
def fun(a,b=0,x):  
    c=a*b*x  
    print(c)  
  
fun(5,3,2)
```

Here we have third argument. Let us see if we get the output as expected, that is $5*3*2 = 30$

Output:

```
File "C:/Users/rooman/OneDrive/Desktop/python/test.py", line 1  
  def fun(a,b=0,x):  
          ^  
SyntaxError: non-default argument follows default argument
```

As the message reflected, non-default argument cannot follow default argument. But certainly after positional arguments we can have as many default arguments as possible.

Keyword arguments

In positional arguments we noticed that if order changes the output also changes. We can overcome this by using keyword arguments, let us see how

The screenshot shows a Jupyter Notebook interface. On the left, a code cell named 'test.py' contains the following Python code:

```
def power_of(a, b):
    c = a**b
    print(c)

power_of(a=2, b=5)
power_of(b=5, a=2)
```

On the right, under the heading 'Keyword arguments', there is a diagram with three boxes labeled 'a', 'b', and 'c' with their corresponding values: 2, 5, and 32.

The 'Output Window' on the left shows the execution of the code. It first runs 'runfile' with arguments 'C:/python/test.py' and 'wdir=C:/python'. The output is '32'. Then it runs another 'runfile' with the same arguments, and the output is again '32'.

We can see that, if we mention the keywords while passing arguments the output does not depend on the order or the position of the arguments. This is the advantage of using keyword arguments.

Variable length argument

Passing any number of arguments to a function is called as variable length argument. Let us know better by an example

The screenshot shows a Jupyter Notebook interface. On the left, a code cell named 'test.py' contains the following Python code:

```
def pizza_toppings(toppings):
    print(toppings)

pizza_toppings("cheese")
pizza_toppings("cheese", "onion", "olives", "corn")
```

On the right, under the heading 'Variable length arguments (Arbitrary arguments)', there is a diagram with four boxes labeled 'cheese', 'onion', 'olives', and 'corn'.

The 'Output Window' on the left shows the execution of the code. It first runs 'runfile' with arguments 'C:/python/test.py' and 'wdir=C:/python'. The output is 'cheese'. Then it runs another 'runfile' with the same arguments, and the output is 'Traceback (most recent call last):'. Below that, it shows the file path 'File "C:\python\test.py", line 5, in <module>' and the error message 'pizza_toppings("cheese", "onion", "olives", "corn")'. At the bottom, it shows the error 'TypeError: pizza_toppings() takes 1 positional argument but 4 were given'.

In the above example we are trying to pass different number of arguments but only first argument is what the function is taking, so certainly some changes in function declaration is needed, let us see what is the change and how that change will accept multiple arguments

Variable length arguments
(Arbitrary arguments)

```
test.py
def pizza_toppings(*toppings):
    print(toppings)

#pizza_toppings("cheese")
pizza_toppings("cheese", "onion", "olives", "corn")
```

Output Window

```
File "C:\python\test.py", line 5, in <module>
    pizza_toppings("cheese", "onion", "olives", "corn")
TypeError: pizza_toppings() takes 1 positional argument but 4 were given

In [4]: runfile('C:/python/test.py', wdir='C:/python')
('cheese', 'onion', 'olives', 'corn')
```

We can now see that by attaching a star or asterisk in front of the parameter, the function is now ready to take different number of arguments. This is because **toppings** is now considered as **tuple**. Let us verify that by printing the type of **toppings**

Variable length arguments
(Arbitrary arguments)

```
test.py
def pizza_toppings(*toppings):
    print(toppings)
    print(type(toppings))

#pizza_toppings("cheese")
pizza_toppings("cheese", "onion", "olives", "corn")
```

Output Window

```
In [4]: runfile('C:/python/test.py', wdir='C:/python')
('cheese', 'onion', 'olives', 'corn')

In [5]: runfile('C:/python/test.py', wdir='C:/python')
('cheese', 'onion', 'olives', 'corn')
<class 'tuple'>
```

The diagram illustrates a tuple as a horizontal sequence of four colored boxes. The first box is light blue and contains the word 'cheese'. The second box is light green and contains the word 'onion'. The third box is light orange and contains the word 'olives'. The fourth box is light purple and contains the word 'corn'. A black curly brace is positioned below the first three boxes, spanning from 'cheese' to 'olives'. An arrow points from the end of the brace to the word 'Tuple' in red text.

Note: * in front of the parameter makes it a tuple. But while calling for it we should just enter name which does not contain *.

Let us see another case

```
def pizza_toppings(*toppings,crust):  
    print(toppings)  
    print(crust)  
  
pizza_toppings("cheese",crust="thin")
```

Output:

```
In [32]: runfile('C:/Users/rooman/OneDrive/Desktop/python/  
test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')  
('cheese',)  
thin
```

In the above example we are trying to pass another argument. But crust is a positional argument/non-default argument. So if we use keyword and assign the value, the output is as expected. But what if we don't use the keyword?? Let's see

```
def pizza_toppings(*toppings,crust):  
    print(toppings)  
    print(crust)  
  
pizza_toppings("cheese","thin")
```

Output:



```
File "C:/Users/rooman/OneDrive/Desktop/python/test.py", line 5,  
in <module>  
    pizza_toppings("cheese","thin")  
  
TypeError: pizza_toppings() missing 1 required keyword-only  
argument: 'crust'
```

Here we see that cheese and thin both are considered as toppings. So if we are passing any arguments before or after the variable length argument it is mandatory to use keyword arguments.

Variable keyword length arguments

In the previous example we saw how to pass different number of arguments, but all of them were **toppings**. What if each argument we pass represents different data. Let us see how to resolve this

```
def collect_student_data(*data):
    print(data)

collect_student_data("Rohit",28,60.5,'M')
```

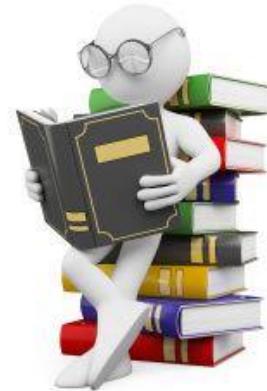
Output:

```
In [34]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
('Rohit', 28, 60.5, 'M')
```

We certainly got the output, but we still can't recognise what is **28, 60.5, M**. To resolve this we must give a key to each value entered as shown below

```
def collect_student_data(*data):
    print(data)

collect_student_data(name="Rohit",age=28,avg=60.5,gender='M')
```



Output:

```
File "C:/Users/rooman/OneDrive/Desktop/python/test.py", line 5,
in <module>
    collect_student_data(name="Rohit",age=28,avg=60.5,gender='M')

TypeError: collect_student_data() got an unexpected keyword
argument 'name'
```

The error states that it doesn't recognise name and similarly age, avg, gender. To resolve this we must make another change, let us see what that is, and how it resolves the issue

```
def collect_student_data(**data):
    print(data)

collect_student_data(name="Rohit", age=28, avg=60.5, gender='M')
```

Output:

```
In [36]: runfile('C:/Users/rooman/OneDrive/Desktop/python/
test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
{'name': 'Rohit', 'age': 28, 'avg': 60.5, 'gender': 'M'}
```



All we did was add another * to the parameter. And we can see in output that the keys and values are enclosed with {} which states ** in front of a parameter makes it a dictionary. Which can store different arguments along with keywords associated with it.

Here also we have the same rule as seen earlier that if we are passing any arguments before or after the variable keyword length argument it is mandatory to use keyword arguments.

Python Fundamentals

day 8

Today's Agenda

- File, script and module
- Built-in modules
- Documentation string



File, script & module

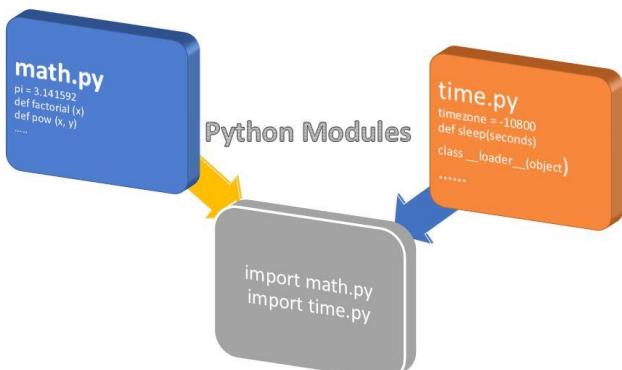
A file from the hard disk when executed on command line then such a program or a file is called as a **script**.

If we have a single file with numerous set of functions then it is difficult to work with such file. The better way is to group similar set of functions and make a separate file of it and name accordingly. These files are called as **modules**.

Whenever we are in need of certain functions, instead of defining a function again we can just use the function present in particular module by bringing to current file. This process is called as

importing and is done by using the keyword **import**.

Importing functions from modules to a file reduces the length of code and certainly reduces the complexity.



Let us now consider an example where we have certain functions in a module called as mymodule and want to use functions present in it, in the python script called as test.py and see how this works.

The screenshot shows a Jupyter Notebook interface. On the left, there is a code cell titled "mymodule.py" containing three function definitions: add(), sub(), and mul(). The "add()" function adds 10 and 20, "sub()" subtracts 10 from 20, and "mul()" multiplies 10 by 20. The right side shows another code cell titled "test.py" which imports "mymodule" and calls its three functions. Below these cells is an "Output Window" containing the results of the execution: 30, 10, and 200 respectively.

```
mymodule.py
def add():
    a = 10
    b = 20
    c = a+b
    print(c)
def sub():
    a = 10
    b = 20
    c = b-a
    print(c)
def mul():
    a = 10
    b = 20
    c = b*a
    print(c)

test.py
import mymodule

mymodule.add()
mymodule.sub()
mymodule.mul()

Output Window
In [2]: runfile('C:/python/test.py', wdir='C:/python')
30
10
200
In [3]:
```

Instead of calling a module by its name every single time we want to use a function from that module, we can rename it and this is called as aliasing. This can be done by using keyword called as **as**. Let us see how to do this

The screenshot shows a Jupyter Notebook interface. On the left, there is a code cell titled "mymodule.py" containing the same three function definitions as before. The right side shows another code cell titled "test.py" which imports "mymodule" and renames it to "mm" using the "as" keyword. It then calls the "add()", "sub()", and "mul()" functions from "mm". Below these cells is an "Output Window" containing the results of the execution: 30, 10, and 200 respectively.

```
mymodule.py
def add():
    a = 10
    b = 20
    c = a+b
    print(c)
def sub():
    a = 10
    b = 20
    c = b-a
    print(c)
def mul():
    a = 10
    b = 20
    c = b*a
    print(c)

test.py
import mymodule as mm

mm.add()
mm.sub()
mm.mul()

Output Window
In [2]: runfile('C:/python/test.py', wdir='C:/python')
30
10
200
In [3]:
```

There is definitely a simpler way for this, which is by using a keyword called as **from**. Let us see how it makes things easy

The screenshot shows a Jupyter Notebook interface. On the left, there is a code cell titled "mymodule.py" containing three functions: add(), sub(), and mul(). The "test.py" cell on the right contains the code "from mymodule import *". In the "Output Window", the command "In [4]: runfile('C:/python/test.py', wdir='C:/python')" is run, followed by "Reloaded modules: mymodule". The output shows the results of calling each function: add() prints 30, sub() prints 10, and mul() prints 200.

```
mymodule.py
def add():
    a = 10
    b = 20
    c = a+b
    print(c)
def sub():
    a = 10
    b = 20
    c = b-a
    print(c)
def mul():
    a = 10
    b = 20
    c = b*a
    print(c)

test.py
from mymodule import *

add()
sub()
mul()

Output Window
In [4]: runfile('C:/python/test.py', wdir='C:/python')
Reloaded modules: mymodule
add()
30
sub()
10
mul()
200
```

In above example ***** means import all the functions present in mymodule to the present file. And we can use the functions just by calling by its name.

If the module contains huge number of functions and we want to use only a couple of functions from it then instead of using ***** and importing all functions we can just mention the function which we want to use and import them selectively as shown below

The screenshot shows a Jupyter Notebook interface. On the left, there is a code cell titled "mymodule.py" containing three functions: add(), sub(), and mul(). The "test.py" cell on the right contains the code "from mymodule import add". In the "Output Window", the command "In [6]: runfile('C:/python/test.py', wdir='C:/python')" is run, followed by "Reloaded modules: mymodule". The output shows the result of calling the imported function add(), which prints 30. A new cell "In [7]" is also visible at the bottom.

```
mymodule.py
def add():
    a = 10
    b = 20
    c = a+b
    print(c)
def sub():
    a = 10
    b = 20
    c = b-a
    print(c)
def mul():
    a = 10
    b = 20
    c = b*a
    print(c)

test.py
from mymodule import add

add()

Output Window
In [6]: runfile('C:/python/test.py', wdir='C:/python')
Reloaded modules: mymodule
add()
30
In [7]:
```

If you want to import more than one function from the module the all you have to do is

The screenshot shows a Jupyter Notebook interface. On the left, the code file `mymodule.py` contains three functions: `add()`, `sub()`, and `mul()`. On the right, the code file `test.py` imports `add` and `sub` from `mymodule`. In the Output Window, running `test.py` outputs the results of `add()` and `mul()`.

```
mymodule.py
def add():
    a = 10
    b = 20
    c = a+b
    print(c)
def sub():
    a = 10
    b = 20
    c = b-a
    print(c)
def mul():
    a = 10
    b = 20
    c = b*a
    print(c)

test.py
from mymodule import add, sub
add()
sub()

Output Window
In [10]: runfile('C:/python/test.py', wdir='C:/python')
Reloaded modules: mymodule
add()
sub()

In [11]:
```

Note: Knowing that a function is present in the module but calling it without importing, will definitely throw an error saying **<function name> is not defined.**

And also do not mix these different ways of importing functions, it will give an error. Better follow just one method.

The screenshot shows a Jupyter Notebook interface. On the left, the code file `mymodule.py` contains three functions: `add()`, `sub()`, and `mul()`. On the right, the code file `test.py` imports `add` and `sub` from `mymodule` and then tries to call `mymodule.mul()`. In the Output Window, running `test.py` results in a traceback error because `mul()` is not imported.

```
mymodule.py
def add():
    a = 10
    b = 20
    c = a+b
    print(c)
def sub():
    a = 10
    b = 20
    c = b-a
    print(c)
def mul():
    a = 10
    b = 20
    c = b*a
    print(c)

test.py
from mymodule import add, sub
add()
sub()
mymodule.mul()

Output Window
In [12]: runfile('C:/python/test.py', wdir='C:/python')
Reloaded modules: mymodule
add()
sub()

Traceback (most recent call last):
  File "C:\python\test.py", line 5, in <module>
    mymodule.mul()
```

Built-in modules

Whatever we have seen so far are user defined modules, where we have created a module with certain functions in it. But there are some modules which contain numerous functions in it and are already present in python these are called as built-in modules.

Let us explore one such called as math

```
In [1]: import math
```

```
In [2]: help(math)
Help on built-in module math:
```

NAME
math

DESCRIPTION
This module provides access to the mathematical functions defined by the C standard.

FUNCTIONS

acos(x, /)
Return the arc cosine (measured in radians) of x.

acosh(x, /)
Return the inverse hyperbolic cosine of x.

asin(x, /)
Return the arc sine (measured in radians) of x.

asinh(x, /)
Return the inverse hyperbolic sine of x.

Activate Windows
Go to Settings to activate Win

help() is a function which will give you complete description about each function present in that module. Above are only few functions present in math module.

And if you just want to know the different functions present in the module without any description the just use **dir()** function as shown below

```
In [3]: dir(math)
Out[3]:
['__doc__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 'acos',
 'acosh',
 'asin',
 'asinh',
 'atan',
 'atan2',
 'atanh',
 'ceil',
 'copysign',
 'cos',
 'cosh',
 'degrees',
 'e',
 'erf',
 'erfc',
 'exp',
```



By looking at the square brackets we can say that it returns the list of functions name present in a particular module.

To get the help for a specific function, let us see what to do

```
In [5]: help(math.factorial)
Help on built-in function factorial in module math:

factorial(x, /)
    Find x!.
    Raise a ValueError if x is negative or non-integral.
```

Let us check if it works as expected

```
In [6]: math.factorial(5)
Out[6]: 120

In [7]: math.factorial(-5)
Traceback (most recent call last):

  File "<ipython-input-7-a46d876612ec>", line 1, in <module>
    math.factorial(-5)

ValueError: factorial() not defined for negative values
```



Let us check some more functions like ceil(), floor()

```
In [8]: help(math.ceil)
Help on built-in function ceil in module math:

ceil(x, /)
    Return the ceiling of x as an Integral.

    This is the smallest integer >= x.
```

For example:

```
In [9]: math.ceil(5)
Out[9]: 5

In [10]: math.ceil(4.45)
Out[10]: 5

In [11]: math.ceil(-11.23)
Out[11]: -11
```

Now let's see what floor() does

```
In [13]: help(math.floor)
Help on built-in function floor in module ma

floor(x, /)
    Return the floor of x as an Integral.

    This is the largest integer <= x.
```

For example:

```
In [14]: math.floor(4.31)
Out[14]: 4

In [15]: math.floor(-12.02)
Out[15]: -13
```



Documentation string

Earlier we saw how to create a module, but we have seen that every module has a description so that it's easy for a programmer to know what does a function inside particular module do. The description provided is called as **documentation**. The definition of that function is called **documentation string** which is enclosed within triple quotes.

Let us understand by an example

The screenshot shows the Python IDLE interface. On the left, there are two code editors: `mymodule.py` and `test.py`. `mymodule.py` contains the following code:

```
def power_of(a,b):
    ''' This function calculates the
    result of a to the power of b '''
    c = a**b
    print(c)

def get_quotient(numerator,denominator):
    ''' This function calculates the
    quotient of numerator divided by denominator '''

    quotient = numerator/denominator
    print(quotient)
```

`test.py` contains:

```
import mymodule
help(mymodule)
```

On the right, the "Output Window" displays the help information for the `mymodule` module:

```
Output Window
Help on module mymodule:
NAME
    mymodule
FUNCTIONS
    get_quotient(numerator, denominator)
        This function calculates the
        quotient of numerator divided by denominator

    power_of(a, b)
        This function calculates the
        result of a to the power of b
FILE
    c:\python\mymodule.py
```

The command `C:\python>_` is shown at the bottom of the output window.

We can certainly see the documentation of all the functions in **power_of** module.

To see description of selected functions, as said earlier we have to use **help(<module_name>.<function_name>)** as shown below

The screenshot shows the Python IDLE interface. On the left, there are two code editors: `mymodule.py` and `test.py`. `mymodule.py` contains the same code as before. `test.py` contains:

```
import mymodule
#help(mymodule)
help(mymodule.power_of)
```

On the right, the "Output Window" displays the help information for the `power_of` function in the `mymodule` module:

```
Output Window
C:\python>python test.py
Help on function power_of in module mymodule:
power_of(a, b)
    This function calculates the
    result of a to the power of b
C:\python>
```

There is also a second way to access the documentation string. Every function has a variable, the name of this variable is `__doc__`(double underscore doc double underscore)/dunder doc. Let us see how to access it

The screenshot shows a Python development environment with two files and their output.

mymodule.py:

```
def power_of(a,b):
    ''' This function calculates the
    result of a to the power of b '''
    c = a**b
    print(c)

def get_quotient(numerator,denominator):
    ''' This function calculates the
    quotient of numerator divided by denominator '''
    quotient = numerator/denominator
    print(quotient)
```

test.py:

```
#help(mymodule.power_of)
#print(mymodule.power_of.__doc__)
print(mymodule.power_of.__doc__)

#help(mymodule.get_quotient)
#print(mymodule.get_quotient.__doc__)
print(mymodule.get_quotient.__doc__)
```

Output Window:

```
C:\python>python test.py
This function calculates the
quotient of numerator divided by denominator
C:\python>
```

The output window shows the results of running `test.py`. It displays the documentation strings for `power_of` and `get_quotient` functions. The `power_of` documentation is shown in a box:

`power_of` *This function calculates the
result of a to the power of b* `_doc_`

The `get_quotient` documentation is shown in a box:

`get_quotient` *This function calculates the
quotient of numerator divided by denominator* `_doc_`

Python Fundamentals

day 9

Today's Agenda

- Return keyword
- Main function
- Taking user inputs



Return keyword

In the other programming languages like Java, C we have seen that

A function can take numerous parameters but return returns only single value. But in python not only does a function accept multiple values as input but also returns multiple values.

```
def fun():
    a=10
    b=20
    c=30
    return a,b,c

print(fun())
```

Output:

```
In [1]: runfile('C:/Users/rooman/OneDrive/Desktop/python/
test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
(10, 20, 30)
```

By looking at the output format we can see that the values are stored in tuple. Which means, when a function returns multiple values it is stored inside tuple.

So now we know how to print. Let us see how to store those values

```
def fun():
    a=10
    b=20
    c=30
    return a,b,c

res=fun()
print(res)
print(type(res))
```

Output:

```
In [2]: runfile('C:/Users/rooman/OneDrive/Desktop/python/
test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
(10, 20, 30)
<class 'tuple'>
```

Now let us see how to store in multiple variables

```
def fun():
    a=10
    b=20
    c=30
    return a,b,c

res1,res2,res3=fun()
print(res1)
print(res2)
print(res3)
```



Output:

```
In [3]: runfile('C:/Users/rooman/OneDrive/Desktop/python/
test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
10
20
30
```

In the above output, the tuple is now unpacked and 10 is given to **res1**, 20 to **res2** and 30 to **res3**.

Note: Number of variables should be equal to number of values returned by the tuple.

Main function – main()

Let us create `mymodule.py` which consists of functions `power_of` and `get_remainder` and then import the module in `test.py`

`mymodule.py`

```
def power_of(a,b):
    """this function calculates the result
    of a raise to the power of b"""
    c=a**b
    print(c)

def get_quotient(numerator,denominator):
    """This function calculates the quotient of
    numerator divided by denominator"""
    quotient=numerator/denominator
    print(quotient)

power_of(2,5)
get_quotient(100,2)
```



`test.py`

```
import mymodule

def get_remainder(num,den):
    '''This function calculates the remainder of num/den'''
    rem = num%den
    print(rem)

get_remainder(100,6)
```

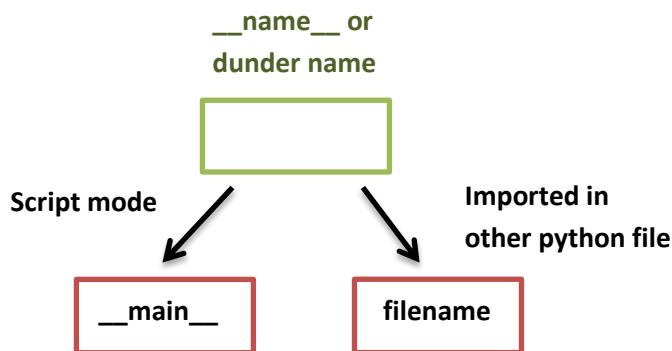
Output:

```
In [5]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
32
50.0
4
```

We can see in the above output, call to the functions from `mymodule` got automatically executed which is not what we wanted.

Whenever we import a module, we just want to import the functions in the module and don't want the function calls to get executed. So let us see how to do that

- Every python file has a variable name called as dunder name. The content inside this variable is based on whether the python file is executed in script mode or imported in other file.
- If it is executed in script mode, dunder name contains `__main__`
- If it is imported in other file, dunder name will be same as the file name.



- Now we know that, the function call commands must execute only when the python file is executed in script mode. So let us make the required changes in `mymodule`

```
def power_of(a,b):
    """this function calculates the result
    of a raise to the power of b"""
    c=a**b
    print(c)

def get_quotient(numerator,denominator):
    """This function calculates the quotient of
    numerator divided by denominator"""
    quotient=numerator/denominator
    print(quotient)

if __name__ == '__main__':
    power_of(2,5)
    get_quotient(100,2)
```

Now if we execute `test.py` we expect function calls to not get executed unless explicitly called for. Let us see if we succeed this time

```
In [6]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
4
```

Certainly we got as expected. Now let us call functions in `mymodule` explicitly in `test.py` and see the output

```
import mymodule

def get_remainder(num,den):
    '''This function calculates the remainder of num/den'''
    rem = num%den
    print(rem)

get_remainder(100,6)
mymodule.power_of(2,5)
mymodule.get_remainder(100,2)
```

Output:

```
In [8]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
4
32
50.0
```

With this information let us create a `main()`

```
def power_of(a,b):
    """this function calculates the result
    of a raise to the power of b"""
    c=a**b
    print(c)

def get_quotient(numerator,denominator):
    """This function calculates the quotient of
    numerator divided by denominator"""
    quotient=numerator/denominator
    print(quotient)

def main():
    power_of(2,5)
    get_quotient(100,2)

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



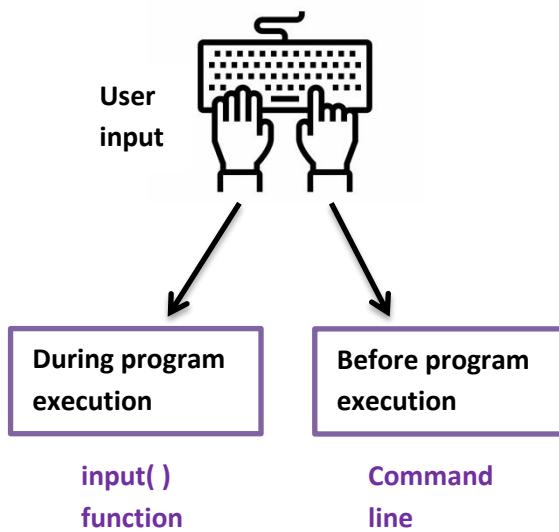
Output:

```
In [9]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
32
50.0
```

Now just like Java, C, C++ here in python also we have **main()** which is where the execution will start.

Taking user inputs

All clients expect the input to be given by user. So it is very important that we know how to take inputs from user.



- **During program execution – `input()`**

```
print("Enter the numerator")
num = input()
print("Enter the denominator")
den = input()

res = num/den
print(res)
```

Output:

```
In [10]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
Enter the numerator

10
Enter the denominator

2
  File "C:/Users/rooman/OneDrive/Desktop/python/test.py", line 6, in <module>
    res = num/den
```



**TypeError: unsupported operand type(s) for /:
'str' and 'str'**

Note: Irrespective of what we enter as input, it is considered as string. As division cannot be performed on strings let us type cast it to integer.

```
print("Enter the numerator")
num = int(input())
print("Enter the denominator")
den = int(input())

res = num/den
print(res)
```

Output:

```
In [11]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
Enter the numerator

10
Enter the denominator

2
5.0
```

We can further more reduce the size of code and make it efficient as following

```
num = int(input("Enter the numerator\n"))
den = int(input("Enter the denominator\n"))

res = num/den
print(res)
```

Output:

```
In [12]: runfile('C:/Users/rooman/OneDrive/  
Desktop/python/test.py', wdir='C:/Users/rooman/  
OneDrive/Desktop/python')
```

```
Enter the numerator  
10
```

```
Enter the denominator  
2  
5.0
```

Before going to the next way of accepting inputs let us see another example

```
exp = input("Enter an expression\n")  
res = eval(exp)  
print(res)
```

eval() is a function which evaluates an expression.

Output:

```
In [13]: runfile('C:/Users/rooman/OneDrive/  
Desktop/python/test.py', wdir='C:/Users/rooman/  
OneDrive/Desktop/python')
```

```
Enter an expression  
3*5-1  
14
```



Python Fundamentals

day 10

Today's Agenda

- Taking input from command line
- print() function
- Flow of control
- Logical operators
- Comparisons



Taking input from command line

Inputs are also called as arguments. Arguments passed to a program via command line, even before the program begins execution is called as **command line arguments**.

These command line arguments are stored in list, and the name of this list is argv. Anything that is typed after python followed by space is considered as input. Which means even the python file name is also considered as input. And as we know list is ordered, which means the first elements with 0th index is always the name of python file.

But if you try to directly access the elements inside the argv list. Compiler throws error, that's because this list is present inside a module called as sys. So without importing that module you cannot access the elements inside the list.

```
test.py
print(argv[0])
print(argv[1])
print(argv[2])

res = argv[1]/argv[2]
print(res) |
```



Output Window

```
C:\python>python test.py 10 2
Traceback (most recent call last):
  File "test.py", line 1, in <module>
    print(argv[0])
NameError: name 'argv' is not defined

C:\python>
```

Let us import sys and see the changes in output

```
test.py
import sys
print(sys.argv[0])
print(sys.argv[1])
print(sys.argv[2])

res = sys.argv[1]/sys.argv[2]
print(res)
Output Window

C:\python>python test.py 10 2
test.py
10
2
Traceback (most recent call last):
  File "test.py", line 7, in <module>
    res = sys.argv[1]/sys.argv[2]
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

The inputs taken through command line are string by default. So we have to change its datatype by type casting before perform any arithmetic operations.

```
test.py
import sys
print(sys.argv[0])
print(sys.argv[1])
print(sys.argv[2])

res = int(sys.argv[1])/int(sys.argv[2])
print(res)
Output Window

C:\python>python test.py 10 2
test.py
10
2
5.0

C:\python>
```

print() function

print() is the function that is used the most while coding. Let us see what is the syntax of it and what are the different arguments a print() expects.

Syntax: `print(value(s), sep=' ', end='\n', file=file, flush=flush)`

Note: Right now let us not concentrate much on the last two arguments file and flush.

Value(s): Accepts multiple input values.

Sep: Separates the values. Separated by spaces(default) if not mentioned any.

End: Ends the line by \n (new line), if not mentioned otherwise.

Let us see some examples



```
In [1]: x=10
In [2]: y=20
In [3]: z=30
In [4]: print(x,y,z)
10 20 30
```

x,y,z are the values here. Separated by **spaces** and ended by **newline**. Which are the default values accepted by the arguments.

Let us try changing the default values for separation of values and ending of the line in the following example

```
In [5]: print(x,y,z,sep='*',end='??')
10*20*30??
In [6]: |
```

We can now see that the values are separated by ***** and line has ended by **??**.

Note: As this is interactive mode we have been directed to new line.
But in case of script mode the cursor would be just after ?? and not
in next line.

```
x=10
y=20
z=30
print(x,y,z)
print(x,y,z,sep='*',end='??')
print("enter something")
```

Output:

```
In [8]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
10 20 30
10*20*30??enter something
```



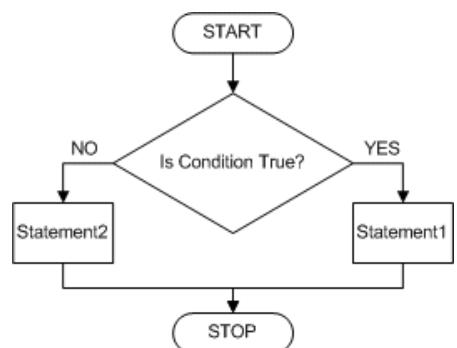
We can see that in script mode second print statement does not end with newline, instead it ends with ?? so as the cursor is after the ?? the next statements gets printed there itself.

Flow of control

The flow of python interpreter is usually sequential. Where it starts from first and executes each line one by one and then ends execution. But there might be certain instances where a set of lines need to be executed only if a certain condition is true/false. As these statements control the flow of program these are called control statements/conditional statements.

We have different conditional statements like:

- ❖ if statement
- ❖ if else statement
- ❖ if elif else statement



if statement

A condition is checked, if it is true the statements under **if** gets executed, if it is false then it comes out of the conditional statement.

```
if True:  
    print("condition is true")
```

Output:

```
In [9]: runfile('C:/Users/rooman/OneDrive/  
Desktop/python/test.py', wdir='C:/Users/rooman/  
OneDrive/Desktop/python')  
condition is true
```



Let us check what happens if the condition is false

```
if False:  
    print("condition is true")
```

Output:

```
In [10]: runfile('C:/Users/rooman/OneDrive/  
Desktop/python/test.py', wdir='C:/Users/rooman/  
OneDrive/Desktop/python')  
  
In [11]: Activate Windows  
Go to Settings to activate Windows.
```

In the above output we can see that nothing gets printed, that's because there is no statement under false condition/else statements which we shall see next.

if else statement

A condition is passed if it is true the statements under **if** gets executed, if it is false then the statements under **else** or false condition gets executed.

Let us see an example of checking if a number is even or odd

```

n=int(input("Enter a number \n"))
if n%2==0:
    print(n,"is even")
else:
    print(n,"is odd")

```

Output:

```
In [11]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
```

```
Enter a number
12
12 is even
```

```
In [12]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
```

```
Enter a number
45
45 is odd
```



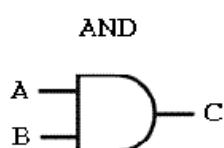
if elif else statement

When there are multiple conditions to be checked we use **if elif else** statement. **Elif** is a short form of **else if**.

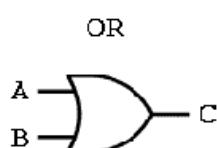
Before knowing this conditional statement with an example let us see different comparisons that can be done and some logical operators

Logical operators

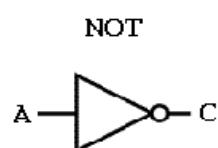
- ❖ and
- ❖ or
- ❖ not



Inputs		Output
A	B	C
0	0	0
0	1	0
1	0	0
1	1	1



Inputs		Output
A	B	C
0	0	0
0	1	1
1	0	1
1	1	1



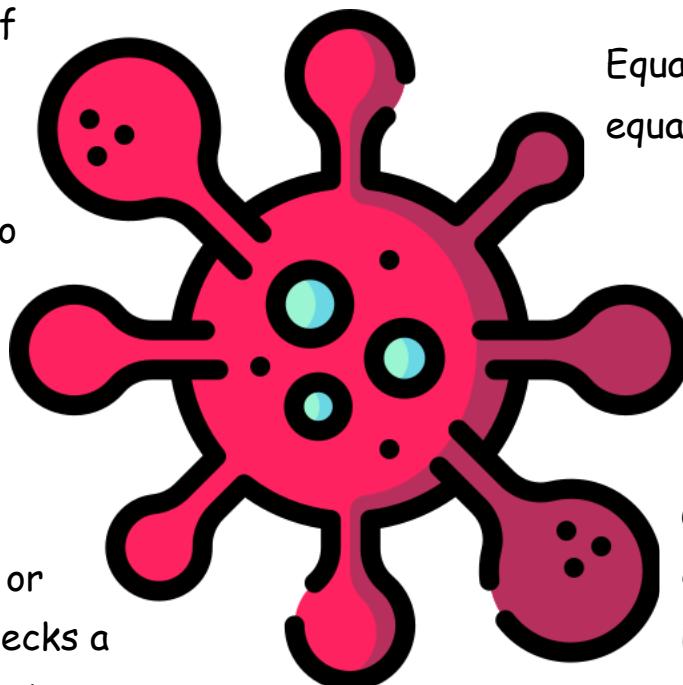
Input	Output
A	C
0	1
1	0

Object identity **is**
checks the equality
of references of
objects.

Comparisons

Less than or equal to
<= checks a number
is lesser than or
equal to the other
number

Greater than or
equal to **>=** checks a
number is greater
than or equal to the
other number

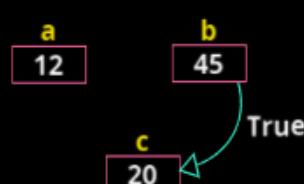


We shall come across many examples where we will be using logical operators and the different comparisons. Let us see an example of **if elif else** statement

```
test.py
a = int(input("Enter 1st number\n"))
b = int(input("Enter 2nd number\n"))
c = int(input("Enter 3rd number\n"))

if a>b and a>c:
    print(a,"is the Largest")
elif b>c:
    print(b,"is the Largest")
else:
    print(c,"is the Largest")

Output Window
Enter 1st number
12
Enter 2nd number
45
Enter 3rd number
20
45 is the largest
```



Truth Table

a	b	a and b
0	0	0
0	1	0
1	0	0
1	1	1

In the above example we have seen **and** operator as well as **if elif else** statement. Now let's see some other logical operators

OR operator

test.py

```
a = 13

if a%3==0 or a%5==0:
    print(a,"is divisible by either 3 or 5")
else:
    print(a,"is not divisible by neither 3 or 5")
```

Truth Table

a	b	a or b
0	0	0
0	1	1
1	0	1
1	1	1

Output Window

```
In [2]: runfile('C:/python/test.py', wdir='C:/python')
10 is divisible by either 3 or 5

In [3]: runfile('C:/python/test.py', wdir='C:/python')
13 is not divisible by neither 3 or 5

In [4]:
```

NOT operator

test.py

```
n = int(input("Enter a number\n"))

if not (n%2==0):
    print(n,"Odd number")
else:
    print(n,"Even number")
```

Truth Table

a	not a
0	1
1	0

Output Window

```
Enter a number
12
12 Even number

In [3]: runfile('C:/python/test.py', wdir='C:/python')

Enter a number
45
45 Odd number
```

In python we have certain values when passed in conditional statements, they evaluate to be false. These values are called **false values** which are: **False, None, 0, empty sequence ("", (), []), empty mapping i.e. {}**

Python Fundamentals

day 11

Today's Agenda

- Looping statements
- While loop
- For loop
- range()
- When to use for and while loop

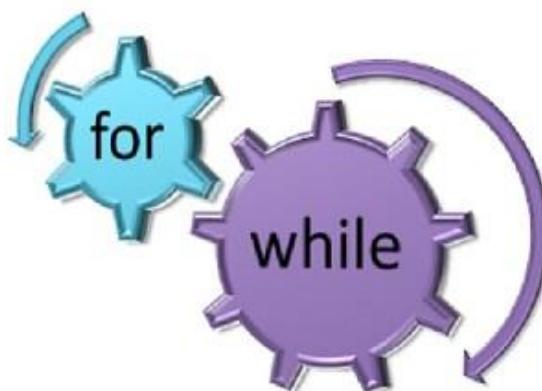


Looping Statements

We can also change the flow of control by using looping statements. Previously we saw how some a part of code gets skipped by not satisfying the condition. Here we shall see how to make a certain section of code repeat itself for given number of times. This is only called as iteration or looping of statements.

In python we have two types of looping statements

- ❖ while loop
- ❖ for loop



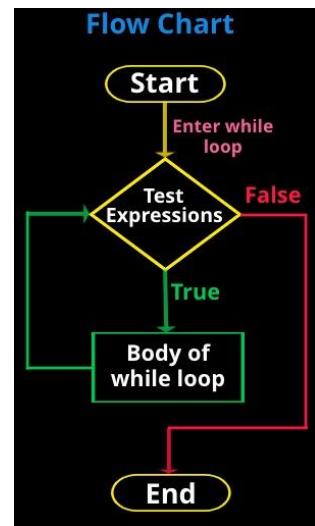
While loop

In **while** loop as long as the condition is satisfied the set of statements repeats its execution. And once the condition is false the control exits the loop and executes the rest of the code if any.

Syntax:

while <condition>:

body of the loop



Let us see an example

```
lst=[10,20,30,40,50]

i=0 #initialisation of variable
while i<5: # condition check
    print(lst[i])
    i=i+1 #incrementation of i
```

Output:

```
In [8]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
10
20
30
40
50
```

Note: initialisation, condition check and incrementation is necessary in while loop.

For loop

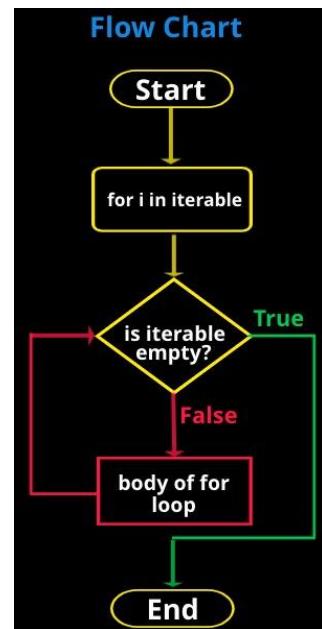
This is less like the `for` keyword in other programming languages, a `for` loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).

Iterable is something which can work on/with `for` loop.

Syntax:

`for var in iterable:`

body of the loop



Let us see an example

```
lst=[10,20,30,40,50]  
for i in lst:  
    print(i)
```

Output:

```
In [9]: runfile('C:/Users/rooman/OneDrive/  
Desktop/python/test.py', wdir='C:/Users/rooman/  
OneDrive/Desktop/python')  
10  
20  
30  
40  
50
```

Above example we see that no condition was checked and no initialisation is required.

range()

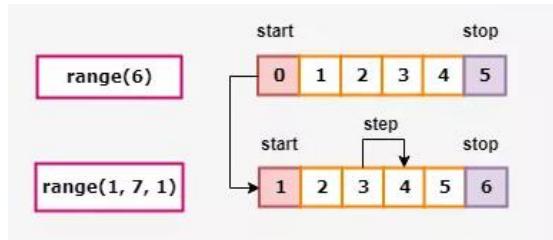
If you want to write for loop in traditional way, that is also possible in python. Where you can specify the number of times your loop should execute by using the function called as `range()`.

Syntax: `range(start, stop, step)`

Start: Which position to start from. By default it is 0. //optional

Stop: Which position to stop at (not included). //required

Step: Number specifying incrementation. Default is 1. //optional



Let us see some examples

```
print(range(5))
```

Output:

```
In [1]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
range(0, 5)
```

Internally a range object is created which is ranging from 0 to 5.

Let us see it in list format

```
print(list(range(5)))
```

Output:

```
In [2]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
[0, 1, 2, 3, 4]
```

Now let us see range starting from non-zero element and with a different step

```
print(list(range(2,10,2)))
```

Output:

```
In [3]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
[2, 4, 6, 8]
```

When to use for and while loop

- ❖ While loop must be used when you are not sure how many times a loop should repeat itself. As long as the condition is true the loop must execute.
- ❖ For loop should be used when you are sure about the number of times the iterations should be performed. Here there is no condition checked.



Let us consider an example of banking where you don't know how many times you have to withdraw the money. As long as the min balance is less than actual balance the withdrawal should continue

```
balance=15500
min_balance=500

print("Balance before transaction:", balance)

while min_balance < balance:
    balance=balance-1000

print("Balance after transaction:", balance)
```

Output:

```
In [6]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
Balance before transaction: 15500
Balance after transaction: 500
```

Above example we don't know how many times the loop should execute, but we know the condition. In such cases **while** loop should be used.

Let's say we know how much to withdraw, considering 5000rs is amount to be withdrawn

```
balance=15500
min_balance=500

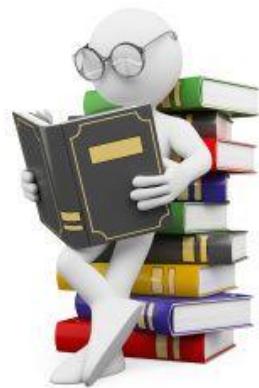
print("Balance before transaction:", balance)

for i in range(5):
    balance=balance-1000

print("Balance after transaction:", balance)
```

Output:

```
In [7]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
Balance before transaction: 15500
Balance after transaction: 10500
```



Above case we know how many times a loop has to execute. And there is no condition to be satisfied. In such cases we use **for** loop.

Let us see the difference between while loop and for loop

while loop	for loop
<ol style="list-style-type: none">1. Initialization of looping variable is required.2. Looping variable need to be incremented/decremened.3. Condition should be checked <p>Use: When we are not sure how many iterations must be performed</p>	<p>Initialization of looping variable is not required.</p> <p>Looping variable need not be incremented/decremened.</p> <p>Condition need not be checked.</p> <p>Use: When we are sure how many iterations must be performed</p>

Python Fundamentals

day 12

Today's Agenda

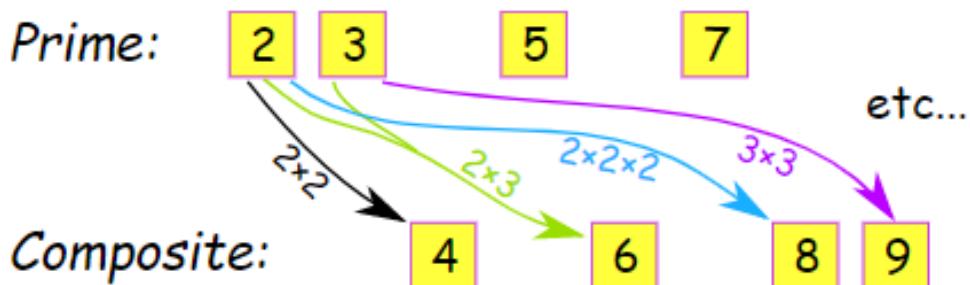
- Break statement
- Continue statement
- Lambda functions



Break statement

Before we know what the break statement does, let us see where and why we should use it. Let us consider an example of prime number

A prime number is a number which is divisible by 1 and the number itself. If a number is divisible by any other number 1 and number itself then the number is not a prime number or also called as composite numbers.



```
n=int(input("Enter a number\n"))

for i in range(2,n+1):
    if n%i==0:
        pass
if i==n:
    print(n,"is prime")
else:
    print(n,"is not prime")
```



Output:

```
In [10]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
```

```
Enter a number
5
5 is prime
```

pass is a null statement. The interpreter does not ignore a **pass** statement, but nothing happens and the statement results into no operation. The **pass** statement is useful when you don't write the implementation of a function but you want to implement it in the future.

Let us see if the same logic works when we enter a non-prime number

```
In [11]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
```

```
Enter a number
6
6 is prime
```

We can see that there is a mistake in logic because 6 is divisible by 2 and 3 before being divisible by number itself therefore it is not a prime number.

So when we check for divisibility of 2, 3 and so on, once we encounter perfect divisibility the current loop should break and resume the next statements. This is done by **break** statement.

Let us see the above example using **break** statement

```
n=int(input("Enter a number\n"))

for i in range(2,n+1):
    if n%i==0:
        break
if i==n:
    print(n,"is prime")
else:
    print(n,"is not prime")
```

Output:

```
In [12]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
```

```
Enter a number
5
5 is prime
```

```
In [13]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
```

```
Enter a number
6
6 is not prime
```

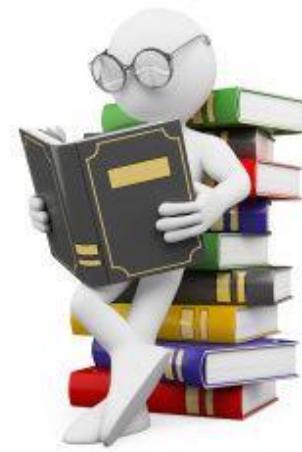
We see that now the logic works perfect for both prime and non-prime numbers.

➤ Let us see some short cut operators with the following example

```
sum=5
print(sum)
sum = sum+5
print(sum)
```

Output:

```
In [14]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
5
10
```



Instead of writing `sum = sum + 5` the shorter version of the same expression is `sum += 5`. Let us verify this

```
sum=5  
print(sum)  
sum +=5  
print(sum)
```

Output:

```
In [15]: runfile('C:/Users/rooman/OneDrive/  
Desktop/python/test.py', wdir='C:/Users/rooman/  
OneDrive/Desktop/python')  
5  
10
```

Same applies for multiplication, division, subtraction etc

Continue statement

Let us first consider an example of calculating even and odd sum

```
even_sum,odd_sum = 0,0  
n=int(input("Enter the value of n:\n"))  
for i in range(1,n+1):  
    if i%2==0:  
        even_sum += i  
  
    odd_sum += i  
  
print("Sum of all even numbers is:",even_sum)  
print("Sum of all odd numbers is:",odd_sum)
```

Output:

```
In [1]: runfile('C:/Users/rooman/OneDrive/  
Desktop/python/test.py', wdir='C:/Users/rooman/  
OneDrive/Desktop/python')  
  
Enter the value of n:  
5  
Sum of all even numbers is: 6  
Sum of all odd numbers is: 15
```



There is some problem with the odd sum logic. That is, once we know the number is even the odd sum statement should not execute, instead `i` value should increment and `if` statement should execute again. This is where `continue` statement would fit in properly.

The **continue** statement rejects all the remaining statements in the current iteration of the loop and moves the control back to the top of the loop. The **continue** statement can be used in both while and for loops.

```
even_sum,odd_sum = 0,0
n=int(input("Enter the value of n:\n"))
for i in range(1,n+1):
    if i%2==0:
        even_sum += i
        continue

    odd_sum += i

print("Sum of all even numbers is:",even_sum)
print("Sum of all odd numbers is:",odd_sum)
```

Output:

```
In [2]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')

Enter the value of n:
5
Sum of all even numbers is: 6
Sum of all odd numbers is: 9
```



Lambda functions

In python whenever we want to create a function we can use two keywords **def** and **lambda**.

We have seen earlier that when we create a function using **def** we have to name the function. Whereas in **lambda** functions there is no name for the function or we can say that it is anonymous function.

Syntax:

lambda arguments : expression

lambda functions are single line, single use or one time use functions.

Let us see how to create one and how to call lambda functions.

```
res = (lambda num,p : num**p) (2,5)
print(res)
```

Output:

```
In [1]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
32
```

What if we want to call the **lambda** function more than once? Is it possible? Because without name how can a function be called? If these are the questions in back of your mind, let us see the next example and get the doubts cleared

```
fun = lambda num,den : num/den
res = fun(100,2)
print(res)

res1 = fun(10,2)
print(res1)
```

Output:

```
In [3]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
50.0
5.0
```



We can see that, to use **lambda** function more than once we have to assign it a reference and pass the inputs using that reference as many times as we want.

Python Fundamentals

day 13

Today's Agenda

- General use of lambda functions
- filter()
- reduce()
- map()
- Other examples



General use of lambda functions

Lambda functions are used whenever the following built-in functions are used

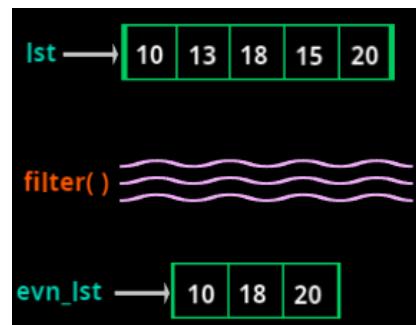
- ❖ filter()
- ❖ reduce()
- ❖ map()

filter()

The **filter()** method filters the given sequence with the help of a function that tests each element in the sequence to be true or not.

Syntax:

filter (function,sequence)



Let us try to understand with the normal function declaration using `def` and then compare with `lambda` function

```
test.py
lst = [10,13,18,15,20]

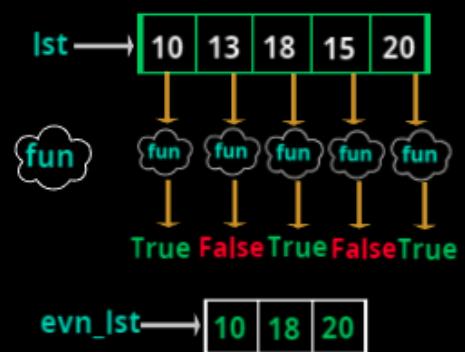
def fun(x):
    if x%2 == 0:
        return True
    else:
        return False

evn_lst = list(filter(fun,lst))
print(evn_lst)
```

Output Window

```
In [3]: runfile('C:/python/test.py', wdir='C:/python')
<filter object at 0x000002486F451D48>
```

```
In [4]: runfile('C:/python/test.py', wdir='C:/python')
[10, 18, 20]
```



To each element in the list `lst`, `fun` function is applied and checks if the result is true or false. All the true resulted elements are collected in a new list `evn_lst` and displayed. In the output we can see that `evn_lst` is a filter object, to get the list we have to type cast it to list. And the new output displays the filtered list `evn_lst`.

Let us now see how the code changes when we use `lambda` function

```
test.py
lst = [10,13,18,15,20]

'''def fun(x):
    if x%2 == 0:
        return True
    else:
        return False

evn_lst = List(filter(fun,lst))
print(evn_lst)'''

result = list(filter(lambda x: (x%2 == 0),lst))
print(result)

Output Window
```

```
In [7]: runfile('C:/python/test.py', wdir='C:/python')
[10, 18, 20]
```

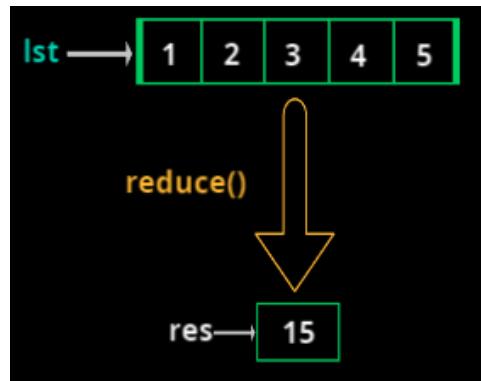
We can see how a **six line code has been reduced to a single line code with less complexity.**

reduce()

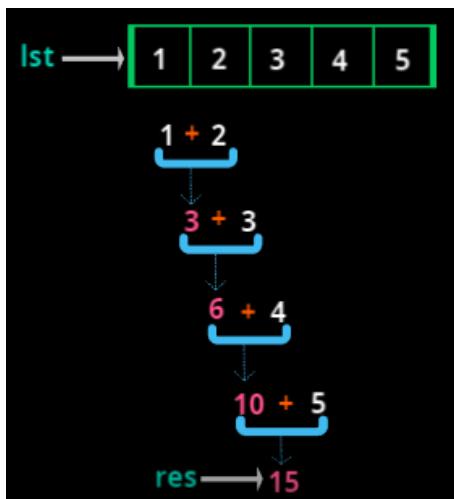
The `reduce()` function is used to apply a particular function passed in its argument to all of the list elements mentioned in the sequence passed along. This function is defined in “`functools`” module.

Syntax:

`reduce (function,sequence)`



In the above diagram we are trying to reduce the list `lst` by adding all the elements and storing the result in `res`.



Let us see how to use `reduce()` with normal `def` function before comparing with `lambda` function

```

test.py
from functools import reduce

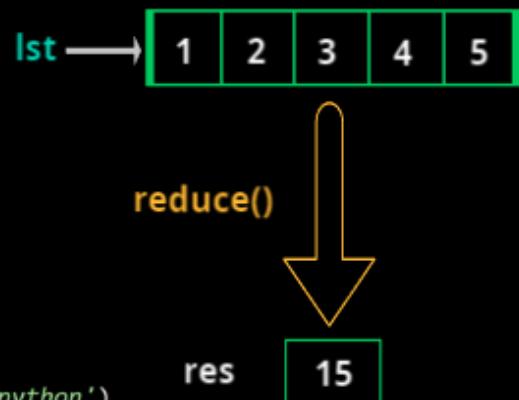
lst = [1,2,3,4,5]

def fun(x,y):
    return x+y

res = reduce(fun,lst)
print(res)

```

Output Window
In [3]: runfile('C:/python/test.py', wdir='C:/python')
15



Using **def** function we had to write the logic in 4 lines let us see how the code changes with the use of **lambda** function

```

test.py
from functools import reduce

lst = [1,2,3,4,5]

'''def fun(x,y):
    return x+y

res = reduce(fun,lst)
print(res)'''

result = reduce(lambda x,y: x+y,lst)
print(result) |

```

Output Window
In [3]: runfile('C:/python/test.py', wdir='C:/python')
15

We can see that the code reduced to a single line statement with less complexity.



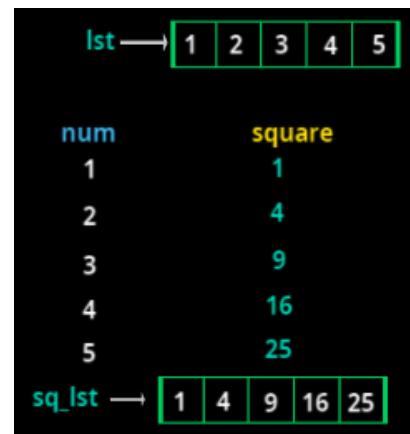
map()

map() function returns a **map** object of the results after applying the given function to each item of a given sequence (list, tuple etc.)

For example let's take a list with certain numbers and map it with its squared values.

Syntax:

map (function,sequence)

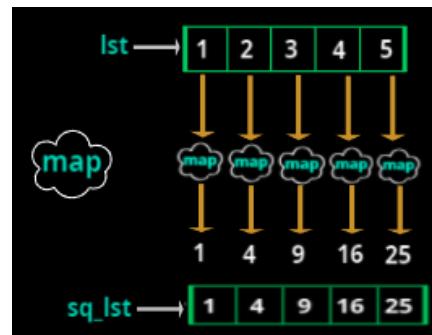


Let us try to code the above logic using **def** function first and then use **lambda** function

```
lst=[1,2,3,4,5]

def fun(x):
    return x**2

sq_lst=list(map(fun,lst))
print(sq_lst)
```



Output:

```
In [6]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
<map object at 0x0000021303701C08>
```

```
In [7]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
[1, 4, 9, 16, 25]
```

In the above output we see that similar to filter, in mapping also map object is created and to see the result in list format we have to type cast it to list.

Let us see the same code with **lambda** function now

```
lst=[1,2,3,4,5]

'''def fun(x):
    return x**2

sq_lst=list(map(fun,lst))
print(sq_lst)'''

sq_lst=list(map(lambda x : x**2,lst))
print(sq_lst)
```

Output:

```
In [8]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
[1, 4, 9, 16, 25]
```



Certainly the code has again reduced with less complexity here as well.

Other examples

In python a function can not only be given as input to a function. If you choose to, you can design a function in such a way that about calling a function it would return a function as output to you.

Let us see how

```
def fun1(num):
    return lambda x : x*num

result=fun1(2)(5)  #(num)(x)
print(result)
```

Output:

```
In [9]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
10
```

But what if we want to call it multiple times?? Let us see the solution

```

def fun1(num):
    return lambda x : x*num

'''result=fun1(2)(5) #(num)(x)
print(result)'''

fun2=fun1(2) #first function call
print(fun2(5)) #call to Lambda function

```



Output:

```

In [10]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
10

```

Like this we can call lambda function multiple times.

Now let us see using `fun1()` how can we create a **mathematical table of any given number**

```

def fun1(num):
    return lambda x : x*num

n= int(input("Enter a number\n"))
math_table=fun1(n)

for i in range(1,11):
    print(n,"X",i,"=",math_table(i))

```

Output:

```

In [11]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')

```

```

Enter a number
7
7 X 1 = 7
7 X 2 = 14
7 X 3 = 21
7 X 4 = 28
7 X 5 = 35
7 X 6 = 42
7 X 7 = 49
7 X 8 = 56
7 X 9 = 63
7 X 10 = 70

```



```
In [12]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
```

Enter a number

```
12  
12 X 1 = 12  
12 X 2 = 24  
12 X 3 = 36  
12 X 4 = 48  
12 X 5 = 60  
12 X 6 = 72  
12 X 7 = 84  
12 X 8 = 96  
12 X 9 = 108  
12 X 10 = 120
```

Similarly we can give any number and generate its mathematical table. Why not try yourself and verify

Multiplication Table

1	2	3	4	5	6
$1 \times 1 = 1$ $1 \times 2 = 2$ $1 \times 3 = 3$ $1 \times 4 = 4$ $1 \times 5 = 5$ $1 \times 6 = 6$ $1 \times 7 = 7$ $1 \times 8 = 8$ $1 \times 9 = 9$ $1 \times 10 = 10$ $1 \times 11 = 11$ $1 \times 12 = 12$	$2 \times 1 = 2$ $2 \times 2 = 4$ $2 \times 3 = 6$ $2 \times 4 = 8$ $2 \times 5 = 10$ $2 \times 6 = 12$ $2 \times 7 = 14$ $2 \times 8 = 16$ $2 \times 9 = 18$ $2 \times 10 = 20$ $2 \times 11 = 22$ $2 \times 12 = 24$	$3 \times 1 = 3$ $3 \times 2 = 6$ $3 \times 3 = 9$ $3 \times 4 = 12$ $3 \times 5 = 15$ $3 \times 6 = 18$ $3 \times 7 = 21$ $3 \times 8 = 24$ $3 \times 9 = 27$ $3 \times 10 = 30$ $3 \times 11 = 33$ $3 \times 12 = 36$	$4 \times 1 = 4$ $4 \times 2 = 8$ $4 \times 3 = 12$ $4 \times 4 = 16$ $4 \times 5 = 20$ $4 \times 6 = 24$ $4 \times 7 = 28$ $4 \times 8 = 32$ $4 \times 9 = 36$ $4 \times 10 = 40$ $4 \times 11 = 44$ $4 \times 12 = 48$	$5 \times 1 = 5$ $5 \times 2 = 10$ $5 \times 3 = 15$ $5 \times 4 = 20$ $5 \times 5 = 25$ $5 \times 6 = 30$ $5 \times 7 = 35$ $5 \times 8 = 40$ $5 \times 9 = 45$ $5 \times 10 = 50$ $5 \times 11 = 55$ $5 \times 12 = 60$	$6 \times 1 = 6$ $6 \times 2 = 12$ $6 \times 3 = 18$ $6 \times 4 = 24$ $6 \times 5 = 30$ $6 \times 6 = 36$ $6 \times 7 = 42$ $6 \times 8 = 48$ $6 \times 9 = 54$ $6 \times 10 = 60$ $6 \times 11 = 66$ $6 \times 12 = 72$
7	8	9	10	11	12
$7 \times 1 = 7$ $7 \times 2 = 14$ $7 \times 3 = 21$ $7 \times 4 = 28$ $7 \times 5 = 35$ $7 \times 6 = 42$ $7 \times 7 = 49$ $7 \times 8 = 56$ $7 \times 9 = 63$ $7 \times 10 = 70$ $7 \times 11 = 77$ $7 \times 12 = 84$	$8 \times 1 = 8$ $8 \times 2 = 16$ $8 \times 3 = 24$ $8 \times 4 = 32$ $8 \times 5 = 40$ $8 \times 6 = 48$ $8 \times 7 = 56$ $8 \times 8 = 64$ $8 \times 9 = 72$ $8 \times 10 = 80$ $8 \times 11 = 88$ $8 \times 12 = 96$	$9 \times 1 = 9$ $9 \times 2 = 18$ $9 \times 3 = 27$ $9 \times 4 = 36$ $9 \times 5 = 45$ $9 \times 6 = 54$ $9 \times 7 = 63$ $9 \times 8 = 72$ $9 \times 9 = 81$ $9 \times 10 = 90$ $9 \times 11 = 99$ $9 \times 12 = 108$	$10 \times 1 = 10$ $10 \times 2 = 20$ $10 \times 3 = 30$ $10 \times 4 = 40$ $10 \times 5 = 50$ $10 \times 6 = 60$ $10 \times 7 = 70$ $10 \times 8 = 80$ $10 \times 9 = 90$ $10 \times 10 = 100$ $10 \times 11 = 110$ $10 \times 12 = 120$	$11 \times 1 = 11$ $11 \times 2 = 22$ $11 \times 3 = 33$ $11 \times 4 = 44$ $11 \times 5 = 55$ $11 \times 6 = 66$ $11 \times 7 = 77$ $11 \times 8 = 88$ $11 \times 9 = 99$ $11 \times 10 = 110$ $11 \times 11 = 121$ $11 \times 12 = 132$	$12 \times 1 = 12$ $12 \times 2 = 24$ $12 \times 3 = 36$ $12 \times 4 = 48$ $12 \times 5 = 60$ $12 \times 6 = 72$ $12 \times 7 = 84$ $12 \times 8 = 96$ $12 \times 9 = 108$ $12 \times 10 = 120$ $12 \times 11 = 132$ $12 \times 12 = 144$

Python Fundamentals

day 14

Today's Agenda

- Scope of variable
- Global variable
- Local variable
- `globals()` & `locals()`
- Recursive function



Scope of variable

A variable is only available from inside the region it is created. This is called **scope**. There are two types of variable

❖ **Global variable with global scope:**

Variable created **outside all functions** are global variables.

These variables can be **accessed through out the code**.

❖ **Local Variable with local scope:**

Variables created **within a function** are local variables. These are **accessed only within that function**.



Global variables

Let us understand using an example

```
x=99 #global variable  
  
def fun():  
    y=999 #Local variable  
    print(y)  
  
fun()
```

Output:

```
In [1]: runfile('C:/Users/rooman/OneDrive/  
Desktop/python/test.py', wdir='C:/Users/rooman/  
OneDrive/Desktop/python')  
999
```



We have declared global and local variable, let us see where these can be accessed

```
x=99 #global variable  
print(x) #in begining of program  
def fun():  
    y=999 #local variable  
    print(y)  
    print(x) #in the function  
  
fun()  
print(x) #at the end of program
```

Output:

```
In [2]: runfile('C:/Users/rooman/OneDrive/  
Desktop/python/test.py', wdir='C:/Users/rooman/  
OneDrive/Desktop/python')  
99  
999  
99  
99
```

In the above example we can see that global variable is accessible throughout the program. Therefore the scope is said to be global.

Local variables

```
1 x=99 #global variable
2
3 def fun():
4     y=999 #local variable
5     print(y)
6
7 fun()
8 print(y) #outside the function
```

Output:



```
File "C:/Users/rooman/OneDrive/Desktop/
python/test.py", line 8, in <module>
    print(y) #outside the function

NameError: name 'y' is not defined
```

In the beginning we saw that **y** can be accessed inside the function. But in the above example when tried to access **y** outside the function we are getting an error that "**y** is not defined". That is because the scope of local variable is restricted inside the function only.

globals() & locals()

Is there a way to know which variable is local and which is global? Certainly there is a way. Let us see what that is

```
test.py
x = 99

def fun():
    y = 999
    print(y)
    print(globals())

fun()
print(x)

Output Window
C:\python>python test.py
999
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <_frozen_importlib_external.SourceFileLoader object at 0x0000028216765748>, '__spec__': None, '__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>, '__file__': 'test.py', '__cached__': None, 'x': 99, 'fun': <function fun at 0x000002B2168B03A8>}
99
```

globals()	
key	value
__name__	__main__
...	...
...	...
...	...
x	99

All the global variables are mapped to their respective values and these are stored in a dictionary. To access this dictionary, you have to call `globals()`. As seen in above output, there are several global variables apart from `x`. For example dunder name, function `fun` is having its object as its value etc.

```
test.py
x = 99

def fun():
    y = 999
    print(y)
    print(globals())
    print(locals())

fun()
print(x)

Output Window
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <_frozen_importlib_external.SourceFileLoader object at 0x0000024580FE5748>, '__spec__': None, '__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>, '__file__': 'test.py', '__cached__': None, 'x': 99, 'fun': <function fun at 0x00000245830103A8>}
[{'y': 999}]
99
```

globals()	
key	value
__name__	__main__
__doc__	...
__package__	...
__loader__	...
__spec__	...
__annotations__	{} (empty)
__builtins__	<module 'builtins' (built-in)>
__file__	'test.py'
__cached__	None
x	99

locals()	
key	value
y	999

Similarly all local variables are mapped to their values and are stored in a dictionary, which can be accessed by calling `locals()`.

A doubt might arise in back of your mind. What if both global and local variable have same name??? Let's see what happens then

```
x=99 #global variable

def fun():
    x=999 #local variable
    print(x)

fun()
```

Output:

```
In [7]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
999
```



Whenever local & global variables have same name, if tried accessing inside a function then local variable is what you'll get.

If at all you want to access global variable inside the function then you have to use a keyword called **global** as shown below

```
x=99 #global variable  
  
def fun():  
    global x  
    x=999 #global variable  
    print(x)  
  
fun()
```

Output:

```
In [9]: runfile('C:/Users/rooman/OneDrive/  
Desktop/python/test.py', wdir='C:/Users/rooman/  
OneDrive/Desktop/python')  
999
```



When we say **global x**, wherever **x** is used it acts like a global variable. To verify this we can print **x** outside the function and cross check the value.

```
x=99 #global variable  
  
def fun():  
    global x  
    x=999 #global variable  
    print(x)  
  
fun()  
print(x)
```

Output:

```
In [10]: runfile('C:/Users/rooman/OneDrive/  
Desktop/python/test.py', wdir='C:/Users/rooman/  
OneDrive/Desktop/python')  
999  
999
```

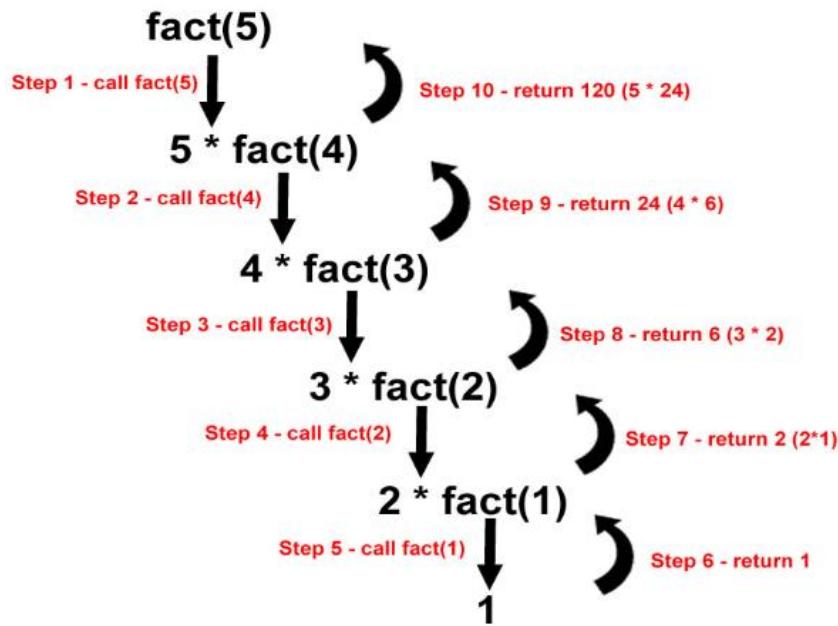


Note that integer is immutable, therefore a new object will get created with new address and the address will be reassigned whenever we are trying to change the values.

Recursive function

In simple words, a function that calls itself is called as recursive function. Let us take a basic example and look further

Considering that we want to calculate the factorial of 5



As we can see to calculate the factorial of a number we have to calculate factorial of its previous number. Which in general can we written as **fact(n) = n*fact(n-1)**. Let us try to code the above logic

```
def fact(n):
    if n==1:
        return 1
    else:
        return n*fact(n-1)

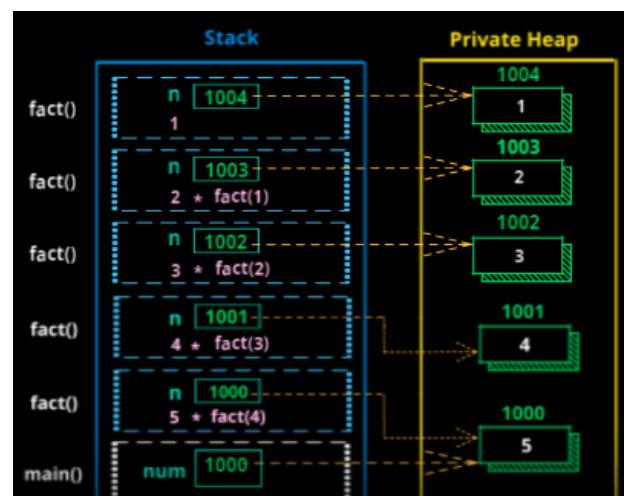
num = int(input("Enter the number:\n"))
print(fact(num))
```

Output:

```
In [12]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
```

```
Enter the number:
5
120
```

Memory perspective



Python fundamentals

day 15

Today's Agenda

- Strings
- Handling a string
- Representation of string



Strings

We have brief knowledge about strings from datatypes section. Let us begin with knowing what are the different ways of creating strings in python

```
s1= '' # an empty string
print(s1)
s2= 'P' # single character string
print(s2)
s3= 'Python' # combination of characters
print(s3)
s4= '''C
Java
Python''' # multiline string
print(s4)
s5= str(99.9) # a real number converted to string
print(s5)
```

For creating string both '' and "" can be used. But for creating a multiline string its mandatory to use """ (triple quotes).

Output:

```
In [2]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')

P
Python
C
Java
Python
99.9
```



Handling a string

We know that a string can be created using '' or ". But what if we want to print a statement like **Practice makes "Perfect"** on the output screen. Let's see what are the different ways of handling such strings in python

```
s='Practice makes perfect'
print(s)
```

Output:

```
In [3]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
Practice makes perfect
```

Great!! But what if we want perfect within quotes?? Let's see further

```
s='Practice makes 'perfect''
print(s)
```

This will certainly give an error. We can predict by noticing the colour change in the string

Considered as a string

'Practice makes 'perfect' ' //perfect is unknown entity for compiler

Considered as a string

Output:

```
File "C:\Users\rooman\Anaconda3\lib\site-
packages\spyder_kernels\customize
\spydercustomize.py", line 110, in execfile
    exec(compile(f.read(), filename, 'exec'),
namespace)

File "C:/Users/rooman/OneDrive/Desktop/
python/test.py", line 1
    s='Practice makes 'perfect''
                           ^
SyntaxError: invalid syntax
```



Let us see what are the different ways to debug this

- **Using escape character (\)**

```
s='Practice makes \'perfect\' # \ is the escape character
print(s)
```

Output:

```
In [5]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
Practice makes 'perfect'
```

Anything that is after the \ is considered to be a part of
string.

- **Using quotes**

```
s='Practice makes "perfect"'
print(s)
```

Output:

```
In [6]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
Practice makes "perfect"
```

If the string is enclosed within '' then we can use " " inside the
string and vice versa.

```
s="Practice makes 'perfect'"
print(s)
```

Output:

```
In [7]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
Practice makes 'perfect'
```

Now that we know how to print a sentence with characters enclosed in '' and ". Let us see what if we want to print "**Practice**" makes '**Perfect**'. Notice that here we are using both '' and " " in one sentence itself

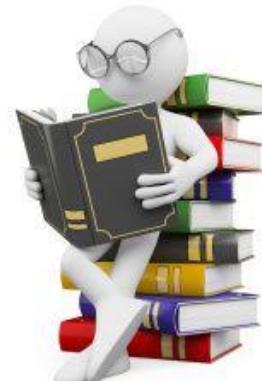
By now we can guess that we certainly cannot enclose the above sentence in '' or ", because that might give an error. So let's try with "" " and see what happens

```
s="""Practice" makes 'perfect"""
print(s)
```

Output:

```
File "C:\Users\rooman\Anaconda3\lib\site-packages\spyder_kernels\customize\spydercustomize.py", line 110, in execfile
    exec(compile(f.read(), filename, 'exec'),
namespace)

File "C:/Users/rooman/OneDrive/Desktop/python/test.py", line 1
    s="""Practice" makes 'perfect"""
                                         ^
SyntaxError: EOL while scanning string literal
```



The error basically means that compiler doesn't know if the string is ending with pair of double quotes ("") or triple quote and a single quote (' ') or single quote and a triple quote (' ""). This can be resolved by giving proper space to separate one from each other

```
s="""Practice" makes 'perfect' """
print(s)
```

Output:

```
In [9]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
"Practice" makes 'perfect'
```



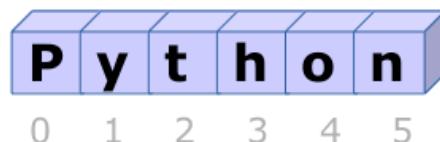
In general outer quotes and inner quotes must not be same.

Representation of string

Let us now see how strings are represented in python

```
s="Python"
print(s)
```

Output:



```
In [10]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
Python
```

When a string is created, a string type object is created on memory. As string is a sequence type object, which means it is ordered and has positional values or index values. When we tell `print(s)` entire string gets printed. But as string is collection of characters we can also access one character at a time. Let us see how

```
s="Python"
print(s)
print(s[4]) #accessing single character
```

Output:

```
In [11]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
Python
o
```



A string is also an iterable in python. Recollect the for loop session, where anything on which a **for** loop can be applied on is called as an iterable.

```
s="Python"  
print(s)  
print(s[4])  
  
for i in s: #passing string s as iterable  
    print(i)
```

Output:

```
In [12]: runfile('C:/Users/rooman/OneDrive/  
Desktop/python/test.py', wdir='C:/Users/rooman/  
OneDrive/Desktop/python')  
Python  
o  
P  
y  
t  
h  
o  
n
```



If you recollect datatypes session, you know that in python strings are immutable. Let us verify this

```
s="Python"  
print(s[4])  
s[4]="i"  
print(s[4])
```

Here we are trying to change the 4th position of string **Python** which is **o** to **i**

Output:

```
File "C:\Users\rooman\Anaconda3\lib\site-  
packages\spyder_kernels\customize  
\spydercustomize.py", line 110, in execfile  
    exec(compile(f.read(), filename, 'exec'),  
namespace)  
  
File "C:/Users/rooman/OneDrive/Desktop/  
python/test.py", line 3, in <module>  
    s[4]="i"  
  
TypeError: 'str' object does not support item  
assignment
```

The above error basically tells us that, in python string values cannot be changed or strings are immutable.

Now let us see some interesting things that strings exhibit

```
s1="hello"
s2="world"
print(s1)
print(s2)
print(id(s1)) #checking the address of s1 object
print(id(s2)) #checking the address of s2 object
print(s1[4])
print(s2[1])
```

Output:

```
In [14]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
hello
world
2429892154032
2429825298736
o
o
```



We see that both the objects `s1` and `s2` are stored in different addresses. Now, both the strings have `l` and `o` in common. Let us see at what address `o` is stored in both the strings

```
s1="hello"
s2="world"
print(s1)
print(s2)
print(id(s1)) #checking the address of s1 object
print(id(s2)) #checking the address of s2 object
print(s1[4])
print(s2[1])
print(id(s1[4])) # address of o in hello
print(id(s2[1])) #address of o in world
```

Output:

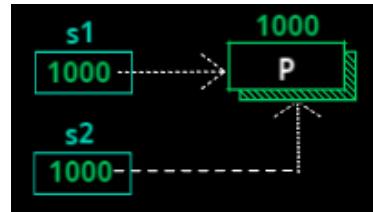
```
In [15]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
hello
world
2429892154032
2429825298736
o
o
2429794338736
2429794338736
```

Interestingly we see that both the o's are stored in same address.
Before knowing how let us take a simple example and understand

```
s1="P"
s2="P"
print(s1)
print(s2)
print(id(s1))
print(id(s2))
```

Output:

```
In [17]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test1.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
P
P
2429796067568
2429796067568
```



As strings are immutable, if the values are same then the address will also be same. Keeping this in back of your mind let us go back to the previous example and know more

Python internally maintains a dictionary, the name of this dictionary is **interned dictionary**. All the single characters are stored as keys and address of those characters are stored as values.

Let us see how exactly that happens

```

s1="hello"
s2="world"
print(s1,s2)
print(id(s1),id(s2)) #address of strings
print(s1[2],s1[3],s2[3]) #all the l's
print(id(s1[2]),id(s1[3]),id(s2[3])) # address of all l's
print(s1[4],s2[1]) #all the o's
print(id(s1[4]),id(s2[1])) #address of all o's

```

Output:

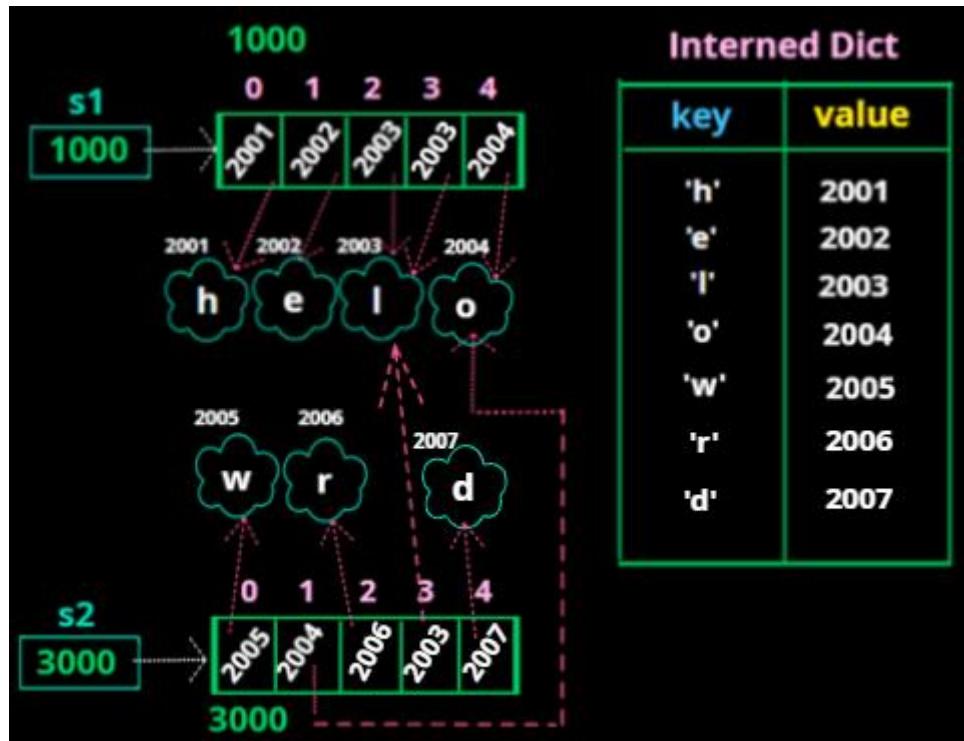
```

In [20]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
hello world
2429891670576 2429825298736
l l l
2429795759920 2429795759920 2429795759920
o o
2429794338736 2429794338736

```



Pictorial representation of string in dictionary format:



Python Fundamentals

day 16

Today's Agenda

- String slicing
- Reversing a string
- Programs on string



String slicing

We have seen how to access single characters from a string. Now let us see how to take portion of a string, taking a portion of string is technically called as slicing.

Considering the string **Guido Van Russom** let us see how slicing can be done

Syntax: inclusive
 ↑
string_name [start : stop+1 : step]
 ↓
 exclusive

You can return a range of characters by using the slice syntax. Specify the start index and the end index, separated by a colon, to return a part of the string.

If not mentioned anything then by default start value is 0, stop value is the length of string + 1 and the step is 1.

```
s = "Guido Van Russom"
print(s)
print(s[10]) #accessing single character
print(s[:]) #Checking the default case
print(s[0:5:]) #slicing the string
```

Output:

```
In [21]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
Guido Van Russom
R
Guido Van Russom
Guido
```



We have seen how to access a single character, a slice of string. Let us now see how step value works

```
s = "Guido Van Russom"
print(s)
print(s[0:9:2]) #slicing the string with step 2
```

Output:

```
In [22]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
Guido Van Russom
GioVn
```

Now let us try slicing the string in reverse way

```
s = "Guido Van Russom"
print(s)
print(s[15:9:-1]) #slicing the string in reverse
```

Output:

```
In [23]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
Guido Van Russom
```



The reason we did not get the string in reverse is because, the step value is 1 which represents forward direction. If we want in reverse direction then step value should be negative.

```
s = "Guido Van Russom"  
print(s)  
print(s[15:9:-1]) #slicing the string in reverse
```

Output:



```
In [24]: runfile('C:/Users/rooman/OneDrive/  
Desktop/python/test.py', wdir='C:/Users/rooman/  
OneDrive/Desktop/python')  
Guido Van Russom  
mossuR
```

In python we also have an additional feature of negative indexes which is much easier while printing a string in reverse.

```
s = "Guido Van Russom"  
print(s)  
print(s[-1:-7:1]) #slicing the string in reverse
```

Output:

```
In [25]: runfile('C:/Users/rooman/OneDrive/  
Desktop/python/test.py', wdir='C:/Users/rooman/  
OneDrive/Desktop/python')  
Guido Van Russom
```

Irrespective of what the start and stop values are, if the step is 1 it always represents forward direction.

```
s = "Guido Van Russom"  
print(s)  
print(s[-1:-7:-1]) #slicing the string in reverse
```

Output:

```
In [26]: runfile('C:/Users/rooman/OneDrive/  
Desktop/python/test.py', wdir='C:/Users/rooman/  
OneDrive/Desktop/python')  
Guido Van Russom  
mossuR
```



Reversing a string

Unlike other programming languages where reversing a string takes few set of lines, in python by knowing all the above techniques we can easily reverse a string.

Let us see how to do it

```
s = "Guido Van Russom"  
print(s)  
print(s[::-1]) #reverse of string
```

Output:

```
In [27]: runfile('C:/Users/rooman/OneDrive/  
Desktop/python/test.py', wdir='C:/Users/rooman/  
OneDrive/Desktop/python')  
Guido Van Russom  
mossuR naV odiuG
```



Note: If we don't mention start and stop values and assign step value as -1. The starting value will be -1 and ending value or stop value will be negative of length of string.

Programs on strings

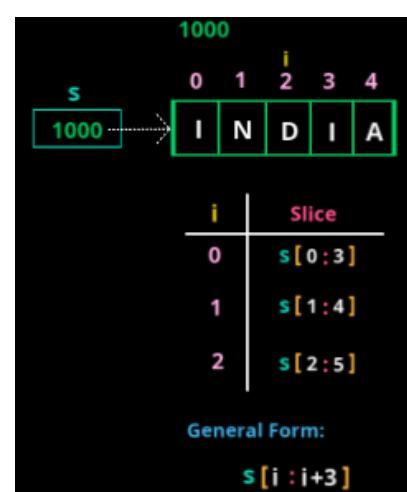
1. Program to print all the substrings with length 3 from the given string

```
s=input("Enter the string:\n")  
  
for i in range(0,len(s)-2):  
    print(s[i:i+3])
```

Output:

```
In [29]: runfile('C:/Users/rooman/OneDrive/  
Desktop/python/test.py', wdir='C:/Users/rooman/  
OneDrive/Desktop/python')
```

```
Enter the string:  
india  
ind  
ndi  
dia
```



You can try the same program with as many strings as you want.

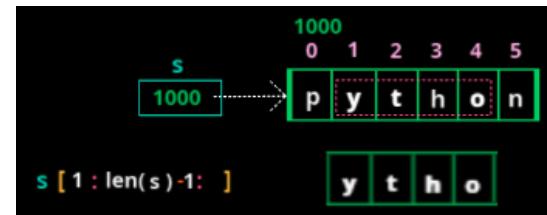
2. Program to print all the characters of a string except the first and the last character.

```
s=input("Enter the string:\n")
print(s[1:len(s)-1:])
```

Output:

```
In [31]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
```

```
Enter the string:
python
ytha
```



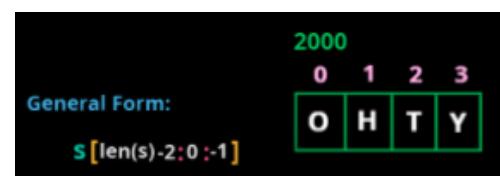
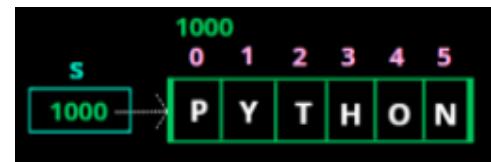
3. Program to print the characters of a string in the reverse order except the first and the last character

```
s=input("Enter the string:\n")
print(s[len(s)-2:0:-1])
```

Output:

```
In [32]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
```

```
Enter the string:
python
htyo
```

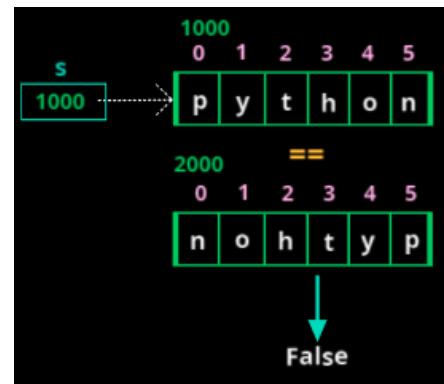
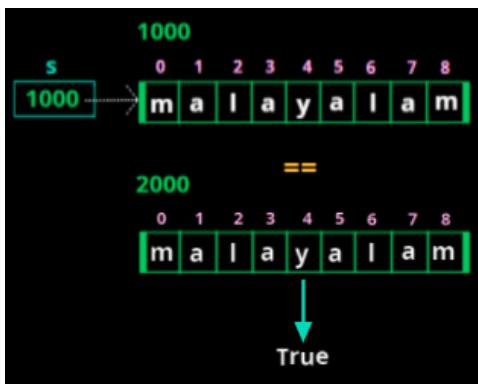


4. Program to check whether the given string is palindrome

```
s=input("Enter the string:\n")

if s==s[::-1]:
    print(s," String is palindrome")
else:
    print(s," String is not a palindrome")
```

Let us try the above with two example : Malayalam, python and see the output



Output:

```
In [33]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
```

Enter the string:
malayalam
malayalam String is palindrome

```
In [34]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
```

Enter the string:
python
python String is not a palindrome

Python fundamentals

day 17

Today's Agenda

- Strings Comparison
- Programs on strings
- String Operations



Strings Comparison

String comparison in python can be performed in three different ways:

- 1) Comparing Values of Strings using `==` operator.
- 2) Comparing References of Strings using `id()` function or `is` operator
- 3) Comparing String values by ignoring their cases using `lower()` and `upper()`.

Let us Understand these different ways of comparing strings in detail with the help of codes as shown below:

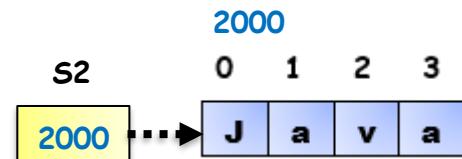


Case I) Comparing Values

Code:

```
s1 = "Python"  
s2 = "Java"
```

```
if s1 == s2:  
    print("String Values are equal")  
else:  
    print("String Values are Unequal")
```



OUTPUT:

String Values are Unequal

Explanation: In the above code we are comparing two strings using == operator which compares the strings based on their values. The two Strings, Python and Java are not equal and hence we get the output as String values are unequal.

Case II) Comparing References

Code:

```
s1 = "Python"  
s2 = "Java"
```

```
if id(s1) == id(s2):  
    print("String references are equal")  
else:  
    print("String references are Unequal")
```



OUTPUT:

String references are Unequal

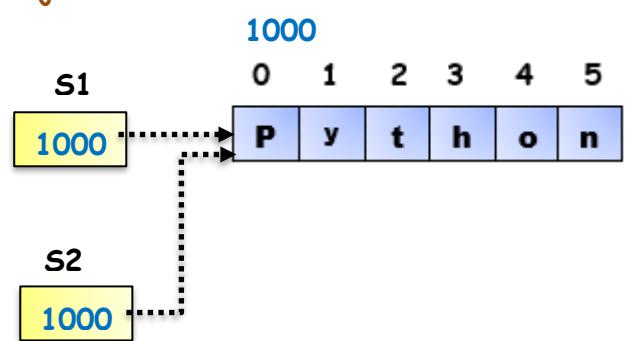
Explanation: In the above code we are comparing two strings References using id() function which compares the address of s1 and s2. S1 has address 1000 and s2 has address 2000 and hence we get the output as String references are unequal.

Case III) Comparing two similar string objects.

Code:

```
s1 = "Python"  
s2 = "Python"
```

```
if s1 == s2:  
    print("String Values are equal")  
else:  
    print("String Values are Unequal")
```



Output:

String Values are equal

Explanation: In the above code we are trying to compare **two similar string objects**. In the first line, when you assign Python to s1, a new string object gets created and address is assigned to it which is then stored in S1. In the second line, you are now trying to create one more string object with the same value as Python.

Will python create one more object with the same value? Think Will it Create?

In order to answer this, one must remember **Strings in python are immutable**, hence two similar string objects are never created in the memory rather they are shared in the memory.



Thus, the same string object is now shared and its address is copied in s2 which now starts pointing to the same python object. When you now compare the values using == operator, we get the output as string values are equal as there is only **one string with two references pointing to it**.

Case IV) Comparing two similar string objects using is operator.

Code:

```
s1 = "Python"  
s2 = "Python"  
  
if s1 is s2:  
    print("String References are equal")  
else:  
    print("String References are Unequal")
```



Output:

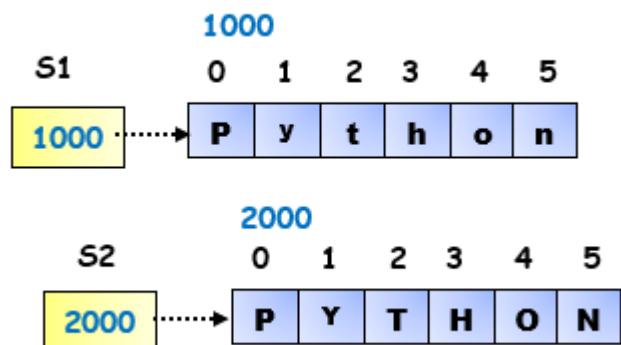
String References are equal

Explanation: In the above code we are trying to compare two string references using **is operator**. Since s1 and s2 are pointing to the same object, they both have the same address which is 1000 and hence we get the output as String references are equal.

Case V) Comparing two similar string objects with different cases.

Code:

```
s1 = "python"  
s2 = "PYTHON"  
  
if s1 == s2:  
    print("String Values are equal")  
else:  
    print("String Values are Unequal")
```



Output:

String Values are Unequal

Explanation: In the above code we are trying to compare two string values using **== operator** which are dissimilar in values as s1 is **python in lower case** and s2 is **PYTHON in upper case** and hence we get the output as String values are unequal.

Programs on strings

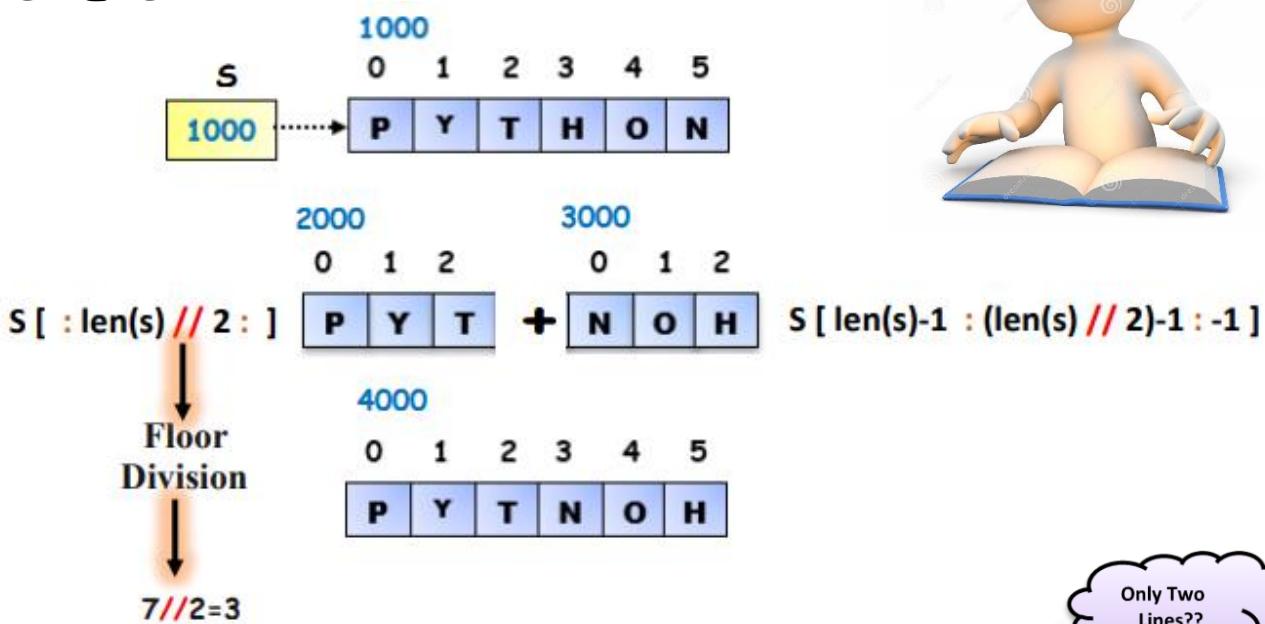
Let us now code some important programs on strings which are commonly asked in many interviews.

1) Write a program to reverse the second half of a string.

Soln: One of the efficient ways of coding the above question is by making use of **Slicing Technique** as shown below.



STEPS:



Only Two Lines??



Code:

```
s = input("Enter a string\n")
print(s[ : len(s)//2 :] + s[len(s)-1 : (len(s)//2)-1 : -1])
```

Output:

```
Enter a string
PYTHON
PYTNOH
```

String Operations

String operations can be performed by making use of **+** and ***** operators.

+ → Performs **Concatenation** when used between two strings.

***** → Performs **Replication** when used between two strings.



STRING CONCATENATION

+ Operator, When applied between two strings does not perform addition instead performs concatenation which simply means it joins the strings and creates a new string.

Didn't get it??? Confused??



Let us try to understand string concatenation with the help of coding examples.

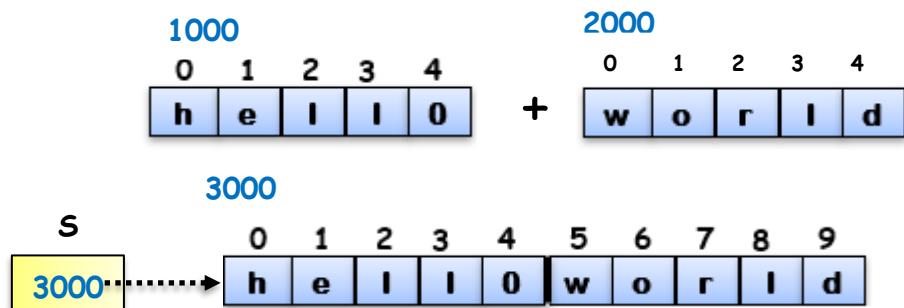
Example1)

Code:

```
s = "hello" + "world"  
print(s)
```

Output:

helloworld



Explanation: In the above example, hello is one string object and world is another string object, when + operator is used between these two strings, the string values does not change as strings are immutable, rather a new string object gets created which consists of the concatenated values of hello and world. It is to this string reference s is assigned. Since hello string object and world string object do not have any references pointing to them (Reference count is zero), they are now treated as garbage objects and are collected by garbage collector



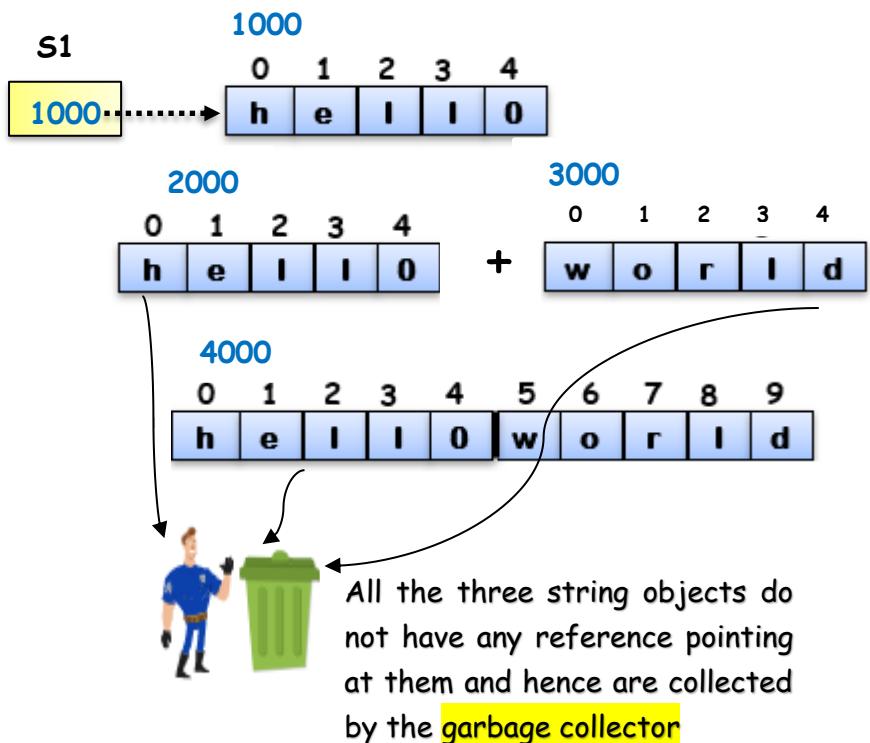
Example2)

Code:

```
s1 = "hello"  
print(s1)  
s1 + "world"  
print(s1)
```

Output:

```
hello  
hello
```



Explanation: In the above example, hello is one string object to which s1 is pointing to. In the third line, we are attempting to change the value of an immutable string by concatenating it with another string called "world" and hence a copy of string object "hello" gets created. To this string we are now trying to modify by concatenating it with world string object. As already discussed, strings cannot be modified, due to which a new string object helloworld gets created to which there is no reference assigned and hence it becomes a garbage object. All the string objects with zero reference count are now collected by the garbage collector and hence the only object in the memory is hello with s1 as reference pointing to it, thus the output we got is hello two times.

Example3)

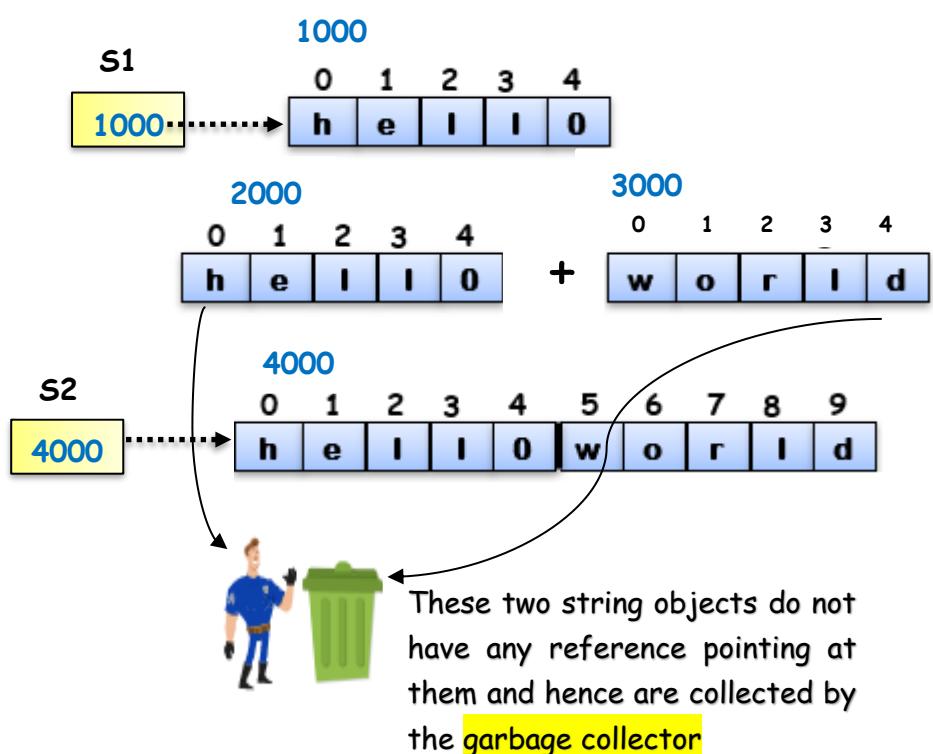
Code:

```
s1 = "hello"  
print(s1)  
s2 = s1 + "world"  
print(s1)  
print(s2)
```



Output

hello
hello
helloworld



Example4)

Code:

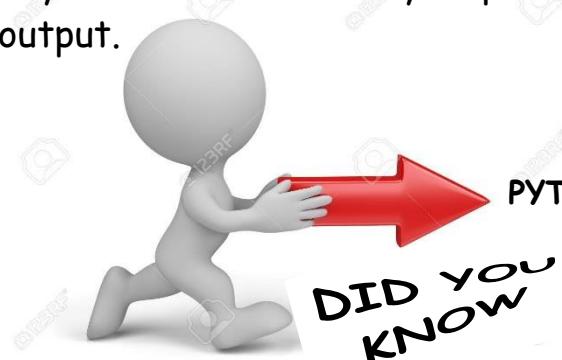
```
s1 = "hello"  
print(s1)  
s1 = s1 + "world"  
print(s1)
```

Output

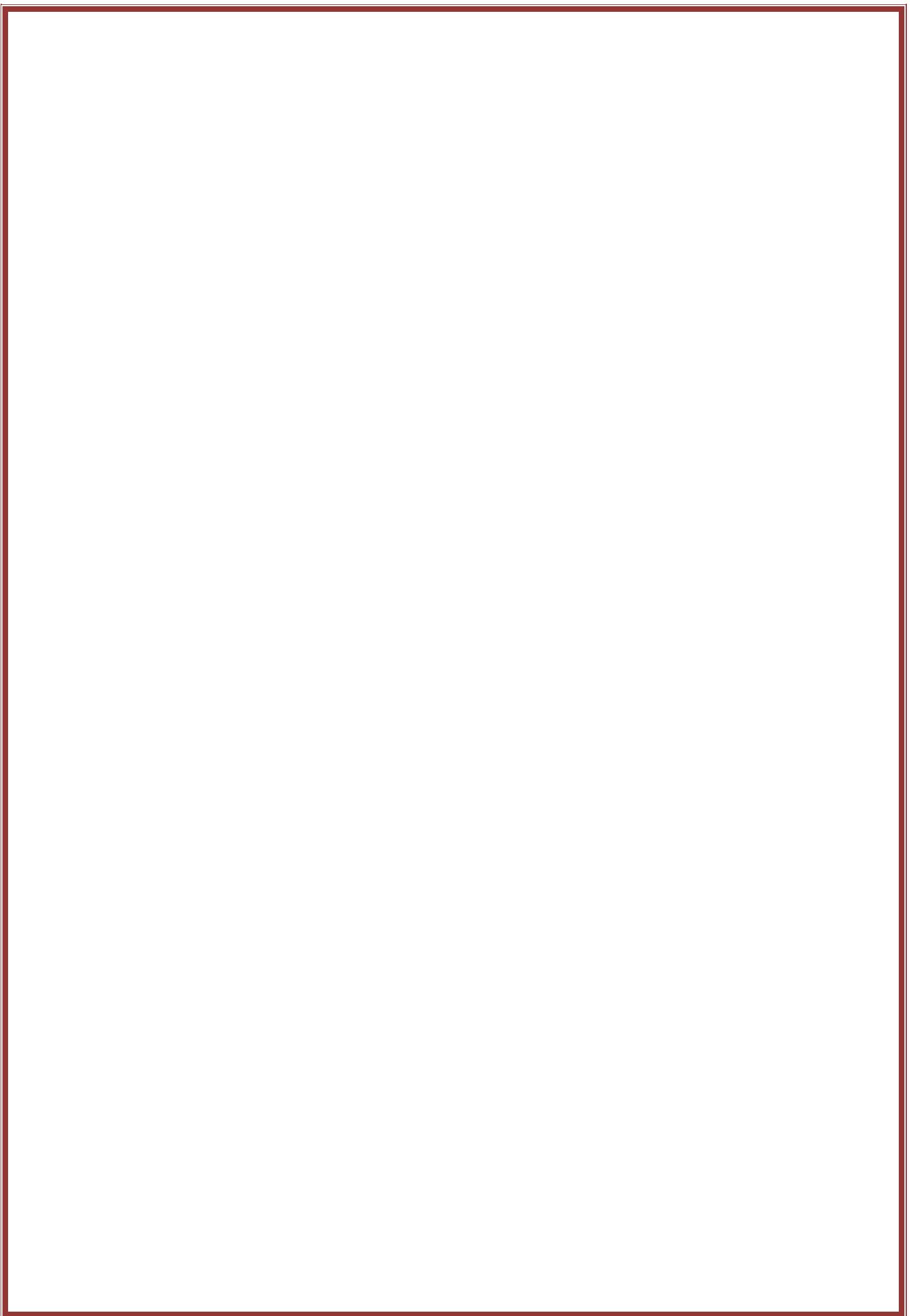
hello
helloworld



Try to draw the memory map for the above code and understand the output.



PYTHON IS AN OPEN SOURCE LANGUAGE.



Python Fundamentals

day 18

Today's Agenda

- String comparison by ignoring cases
- Ascii Table
- Conversion to lowercase
- upper() and lower()
- join()



String Comparison by ignoring cases

We had seen different cases of comparing two strings except the above case. Let us now learn how to compare two strings by ignoring their cases.

In order to compare two strings without considering their cases, one must understand **ASCII table** as python is case sensitive.

ASCII is the acronym for

the American Standard Code for Information Interchange.

It is a code for representing **128**

English characters as numbers, with each letter assigned a **number from 0 to 127**.

For example, the ASCII code for uppercase M is 77 and lowercase m is 109. Most computers use ASCII codes to represent text, which makes it possible to transfer data from one computer to another.

Using the range of lowercase alphabets i.e 97 to 122, we can check whether it is a uppercase or lowercase alphabet and perform the conversion.



ASCII TABLE

Have a look at the ASCII TABLE shown below:

Dec	Hex	Binary	Char	Description	Dec	Hex	Binary	Char	Dec	Hex	Binary	Char	Dec	Hex	Binary	Char
0	0	0000 0000	NUL	Null character	32	20	0010 0000	space	64	40	0100 0000	@	96	60	0110 0000	`
1	1	0000 0001	SOH	Start of Heading	33	21	0010 0001	!	65	41	0100 0001	A	97	61	0110 0001	a
2	2	0000 0010	STX	Start of Text	34	22	0010 0010	"	66	42	0100 0010	B	98	62	0110 0010	b
3	3	0000 0011	ETX	End of Text	35	23	0010 0011	#	67	43	0100 0011	C	99	63	0110 0011	c
4	4	0000 0100	EOT	End of Tx	36	24	0010 0100	\$	68	44	0100 0100	D	100	64	0110 0100	d
5	5	0000 0101	ENQ	Enquiry	37	25	0010 0101	%	69	45	0100 0101	E	101	65	0110 0101	e
6	6	0000 0110	ACK	Acknowledgement	38	26	0010 0110	&	70	46	0100 0110	F	102	66	0110 0110	f
7	7	0000 0111	BEL	Bell	39	27	0010 0111	'	71	47	0100 0111	G	103	67	0110 0111	g
8	8	0000 1000	BS	Backspace	40	28	0010 1000	(72	48	0100 1000	H	104	68	0110 1000	h
9	9	0000 1001	HT	Horizontal Tab	41	29	0010 1001)	73	49	0100 1001	I	105	69	0110 1001	i
10	A	0000 1010	LF	Line Feed	42	2A	0010 1010	*	74	4A	0100 1010	J	106	6A	0110 1010	j
11	B	0000 1011	VT	Vertical Tab	43	2B	0010 1011	+	75	4B	0100 1011	K	107	6B	0110 1011	k
12	C	0000 1100	FF	Form Feed	44	2C	0010 1100	,	76	4C	0100 1100	L	108	6C	0110 1100	l
13	D	0000 1101	CR	Carriage Return	45	2D	0010 1101	-	77	4D	0100 1101	M	109	6D	0110 1101	m
14	E	0000 1110	SO	Shift Out	46	2E	0010 1110	.	78	4E	0100 1110	N	110	6E	0110 1110	n
15	F	0000 1111	SI	Shift In	47	2F	0010 1111	/	79	4F	0100 1111	O	111	6F	0110 1111	o
16	10	0001 0000	DLE	Data Link Escape	48	30	0011 0000	0	80	50	0101 0000	P	112	70	0111 0000	p
17	11	0001 0001	DC1	Device Control 1	49	31	0011 0001	1	81	51	0101 0001	Q	113	71	0111 0001	q
18	12	0001 0010	DC2	Device Control 2	50	32	0011 0010	2	82	52	0101 0010	R	114	72	0111 0010	r
19	13	0001 0011	DC3	Device Control 3	51	33	0011 0011	3	83	53	0101 0011	S	115	73	0111 0011	s
20	14	0001 0100	DC4	Device Control 4	52	34	0011 0100	4	84	54	0101 0100	T	116	74	0111 0100	t
21	15	0001 0101	NAK	Negative ACK	53	35	0011 0101	5	85	55	0101 0101	U	117	75	0111 0101	u
22	16	0001 0110	SYN	Synchronous Idle	54	36	0011 0110	6	86	56	0101 0110	V	118	76	0111 0110	v
23	17	0001 0111	ETB	End of Tx Block	55	37	0011 0111	7	87	57	0101 0111	W	119	77	0111 0111	w
24	18	0001 1000	CAN	Cancel	56	38	0011 1000	8	88	58	0101 1000	X	120	78	0111 1000	x
25	19	0001 1001	EM	End of Medium	57	39	0011 1001	9	89	59	0101 1001	Y	121	79	0111 1001	y
26	1A	0001 1010	SUB	Substitute	58	3A	0011 1010	:	90	5A	0101 1010	Z	122	7A	0111 1010	z
27	1B	0001 1011	ESC	Escape	59	3B	0011 1011	;	91	5B	0101 1011	[123	7B	0111 1011	{
28	1C	0001 1100	FS	File Separator	60	3C	0011 1100	<	92	5C	0101 1100	\	124	7C	0111 1100	
29	1D	0001 1101	GS	Group Separator	61	3D	0011 1101	=	93	5D	0101 1101]	125	7D	0111 1101	}
30	1E	0001 1110	RS	Record Separator	62	3E	0011 1110	>	94	5E	0101 1110	^	126	7E	0111 1110	~
31	1F	0001 1111	US	Unit Separator	63	3F	0011 1111	?	95	5F	0101 1111	_	127	7F	0111 1111	DEL

Let us understand the conversion from uppercase to lowercase and vice-versa.

Consider characters a and A whose ascii values are 97 and 65.

Simple addition and subtraction will help us perform the conversion.

If we subtract 32 from 97, we get 65 which is the ascii value of A.

Char	Value
a	97
↓	-
A	65



Note: Uppercase alphabets fall in the range of 65 to 90
Lowercase alphabets fall in the range of 97 to 122.



String Conversion to lowercase

Let us understand the above conversion with the help of codes:

CODE:

```
c = "a"  
print(ord(c)) #ord() gives ascii value of a stored in c  
print(ord(c) - 32) # performs 97(ascii value of a) - 32 and gives 65  
print(chr(ord(c) - 32)) # gives character whose ascii value is 65
```

OUTPUT:

```
97  
65  
A
```

Explanation: In the above code we used two functions `ord()` & `chr()`.

`ord()` converts a character to an integer and gives its ascii value.

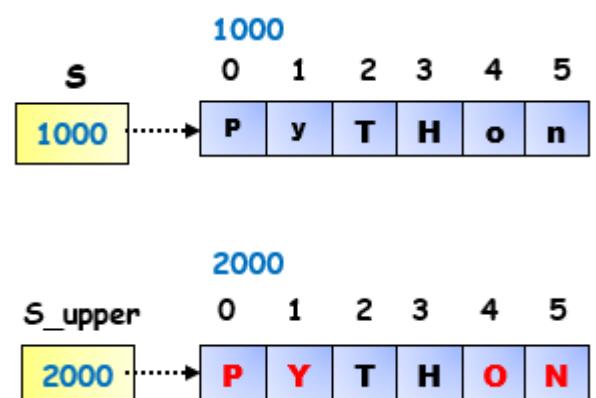
`chr()` converts an integer to a character based on its ascii value.

Above code performs the conversion of lowercase a to uppercase A.

Let us now try writing logic for the conversion of an entire string to Uppercase.

CODE:

```
s = input("Enter a String\n")  
s_upper = ""  
for i in s:  
    if ord(i) >= 97 and ord(i) <= 122:  
        s_upper += chr(ord(i)-32)  
    else:  
        s_upper += i  
  
print(s)  
print(s_upper)
```



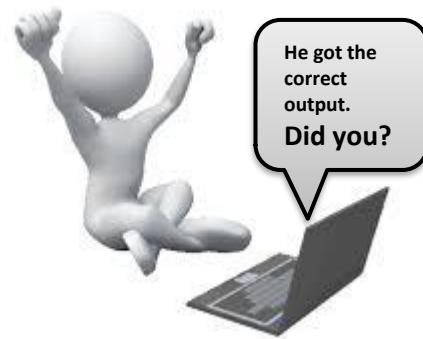
OUTPUT:

Enter a String

pyTHon

pyTHon

PYTHON



Explanation: In the above code we are first checking if each character inside the entered string falls in the range of (97,122) which simply means it is a lowercase character if true. If the condition gets evaluated to true, we performing the conversion from **lowercase to uppercase by subtracting 32** to the ascii value of character and storing it in a new string which **is s_upper**. If in case condition fails, then the character inside string is not a lowercase and do not require any conversion and hence we are simply appending the original character from **string s to string s_upper**. Thus, by **subtracting 32 to any lowercase character we get the ascii value of its uppercase character.**

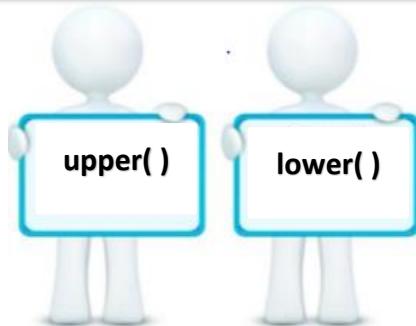
The output we got using the above lines of code can also be achieved in just one line by calling a **built-in function upper()**.

Also vice-versa can be performed by making use of built-in function **lower()**.

upper() and lower()

upper() function on a string converts all characters to uppercase.

lower() function on a string converts all characters to lowercase().



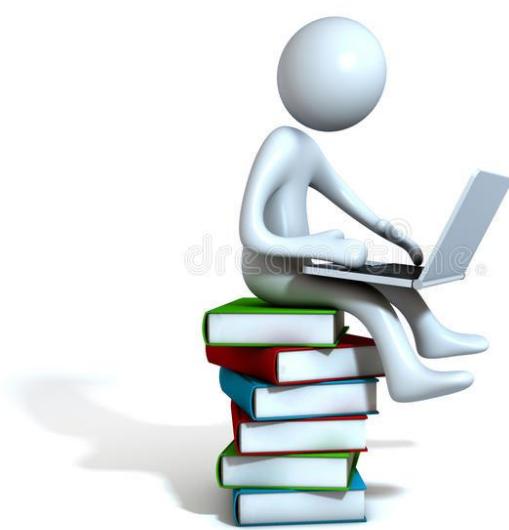
CODE:

```
s = input("Enter a String\n")
s_lower = s.lower()

print(s)
print(s_lower)
```

OUTPUT:

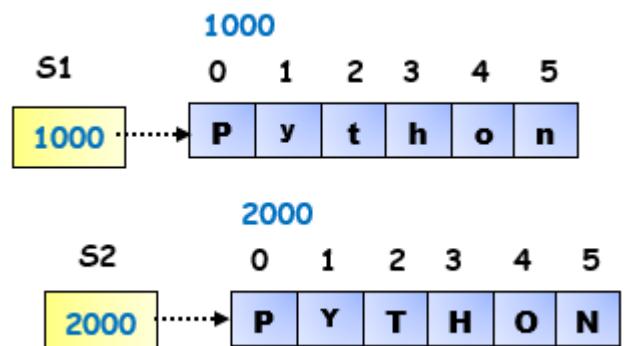
```
Enter a String
PYTHON
PYTHON
python
```



After getting to know all this, let us now compare two strings by ignoring their cases.

CODE:

```
s1 = "python"
s2 = "PYTHON"
if s1.upper() == s2.upper():
    print("String values are equal")
else:
    print("String values are unequal")
```



CODE:

```
s1 = "python"
s2 = "PYTHON"
if s1.lower() == s2.lower():
    print("String values are equal")
else:
    print("String values are unequal")
```



OUTPUT:

```
String Values are equal
```

upper()

We have seen how to perform concatenation between strings using + operator but this approach is not efficient when concatenation between multiple strings has to be performed.

Why you wonder???



Let us understand this with the help of code.

CODE:

```
lst = ["Python" , "Java" , "Django" , "Spring"]
s = ""
for i in lst:
    s += i
print(s)
```

#38535096

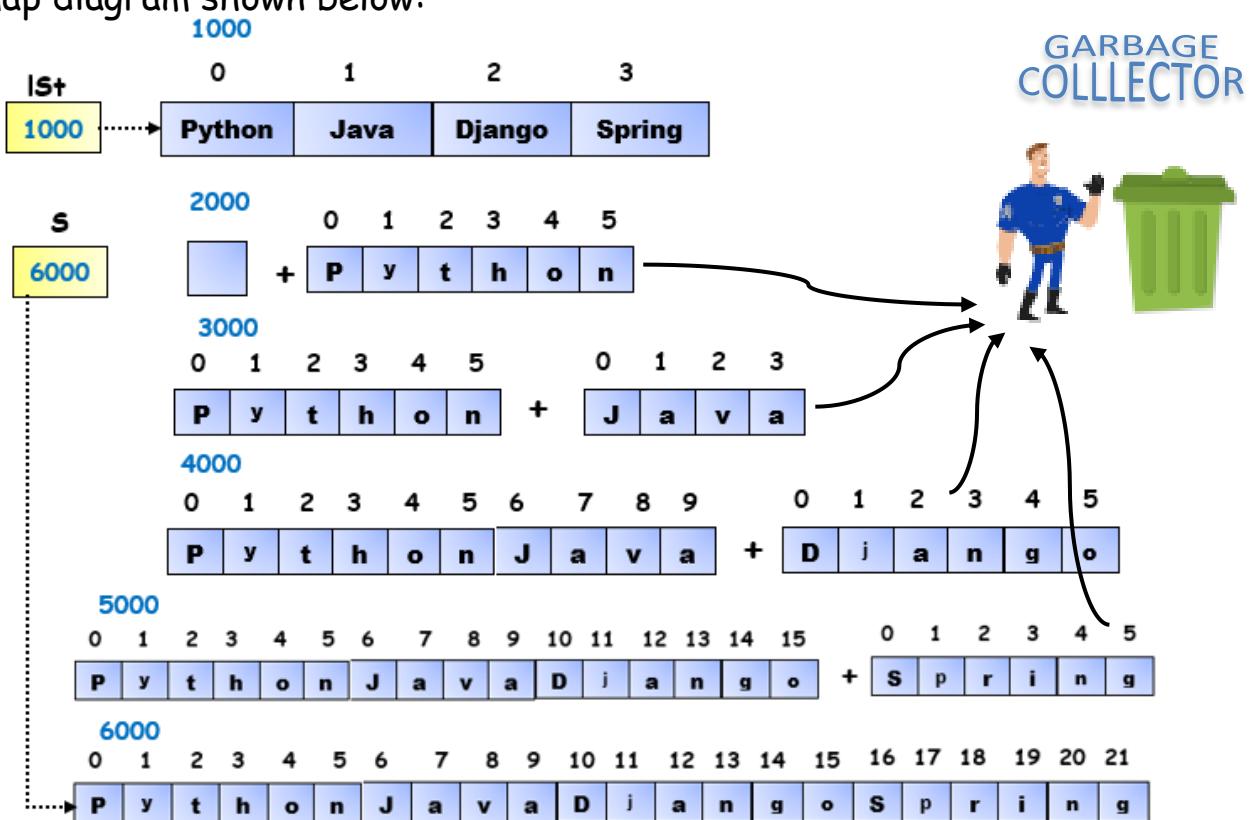
OUTPUT:

PythonJavaDjangoSpring

Explanation: In the above code, we have first created a list which now gets allocated memory with some address and has four string values, Python, Java, Django, Spring with lst as reference pointing to it. In the next line we are creating an empty string object with s as reference to it. Inside the for loop to the empty string s, we are attaching the value of i. The first value of i is Python and when concatenation happens, a new string object gets created which consists of added result of an empty string and Python. We are now storing it back in s which means s is now pointing to this newly created string object with address 3000 and hence the previous objects to which s was pointing to, will become garbage object and is collected by garbage Collector. In the second iteration, i takes the value Java which is now concatenated with Python String object having address i 3000. Since we are performing concatenation, a new string object gets created having concatenated result of Python and java with

address 4000. The added result is now stored back into `s` and hence `s` is now pointing to this string object whose address is 4000. In the next iteration, `i` gets the value `Django` which is now concatenated with `PythonJava`. Since we are performing concatenation, a new string object gets created which consists of added result of `PythonJavaDjango` with 5000 as address. Similarly, in the last iteration `i` gets the value `Spring` which is now concatenated with `PythonJavaDjangoSpring`. Since we are performing concatenation, a new string object gets created which consists of added result of `PythonJavaDjangoSpring` with 6000 as address and this value is stored back into `s` which means `s` is now pointing to this string object with address 6000 and all the other objects who do not have any reference become garbage object as their reference count is zero and hence, are collected by garbage collector.

This way, when we use `+` operator to perform multiple concatenation operations, memory is not utilised efficiently as multiple string objects are allocated and deallocated memory multiple times and memory gets wasted which directly affects the performance of the software. Understand the above scenario with the help of memory map diagram shown below:



Wondering, is there any better approach to concatenate multiple strings?



The better approach is using **join()** function.

join() in python joins each element of an iterable (Such as list, tuple and strings) and returns the concatenated string.

CODE:

```
lst = ["Python" , "Java" , "Django" , "Spring"]
s = "".join(lst)

print(s)
```

OUTPUT:

PythonJavaDjangoSpring



Explanation: join() in the above code takes all the elements present in the list and joins them into one string. Have a look at the memory map diagram shown below where in **one shot join() concatenates all the strings and stores them into s.**



Python fundamentals

day 19

Today's Agenda

- Strings built-in functions (contd)
- Programs on strings
- String translation



Strings built-in functions (contd)

To continue with the built-in functions, let us see the below example where we shall be checking if the string starts with **https**, if yes then print the string

```
url = ["https://www.google.com/", "https://www.youtube.com/"
       , "http://www.xyz.com", "http://www.abc.com"]

for i in url:
    if i[0:5:] == "https":
        print(i)
```

Output:

```
In [35]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
https://www.google.com/
https://www.youtube.com/
```

In the above example we are using string slicing method, now let us see can we do the same using built-in functions

```
url = ["https://www.google.com/", "https://www.youtube.com/"
       , "http://www.xyz.com", "http://www.abc.com"]

for i in url:
    if i.startswith("https"):
        print(i)
```

Output:

```
In [36]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
https://www.google.com/
https://www.youtube.com/
```



Now let us see a similar example but now we shall check how many amongst the `url` list ends with `com`

```
url = ["https://www.google.com/", "https://www.youtube.com/"
       , "http://www.xyz.com", "http://www.abc.org"]

for i in url:
    if i[len(i)-3::] == "com" or i[len(i)-4::] == "com/":
        print(i)
```

Output:

```
In [39]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
https://www.google.com/
https://www.youtube.com/
http://www.xyz.com
```



Let us see is there any built-in function which can do the same function with reduction of code complexity

```
url = ["https://www.google.com/", "https://www.youtube.com/"
       , "http://www.xyz.com", "http://www.abc.org"]

for i in url:
    if i.endswith("com") or i.endswith("com/"):
        print(i)
```

Output:

```
In [40]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
https://www.google.com/
https://www.youtube.com/
http://www.xyz.com
```

We can simplify the above example like below

```
url = ["https://www.google.com/", "https://www.youtube.com/"  
       , "http://www.xyz.com", "http://www.abc.org"]  
  
for i in url:  
    if i.endswith(("com", "com/")): #by giving two conditions as a tuple format  
        print(i)
```

Note: If we are passing multiple inputs to `endswith()` then it should be passed as tuple.

Output:

```
In [41]: runfile('C:/Users/rooman/OneDrive/  
Desktop/python/test.py', wdir='C:/Users/rooman/  
OneDrive/Desktop/python')  
https://www.google.com/  
https://www.youtube.com/  
http://www.xyz.com
```

Programs on strings

1. Program to count number of lower case, upper case, numbers and special characters present in a string.

```
s=input("Enter the string:")  
low_count,up_count,sp_count,num_count=0,0,0  
  
for i in s:  
    if i.islower():  
        low_count += 1  
    elif i.isupper():  
        up_count += 1  
    elif i.isnumeric():  
        num_count += 1  
    else:  
        sp_count += 1  
  
print("Lower case=",low_count)  
print("Upper case=",up_count)  
print("Numeric=",num_count)  
print("Special character=",sp_count)
```



Output:

```
In [42]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
```

```
Enter the string:PyThOn@123
Lower case= 3
Upper case= 3
Numeric= 3
Special character= 1
```

UPPER	lower
1234567	@_#\$^&

Now why don't you try the same example without using any built-in method.

2. Program to swap the case. That is, change upper case to lower and vice versa using `swapcase()`.

```
s="STAY HOME stay safe"
s1=s.swapcase()
print(s)
print(s1)
```

Output:

```
In [43]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
STAY HOME stay safe
stay home STAY SAFE
```



3. Using the same above example let's see what `title()` and `capitalize()` do

```
s="STAY HOME stay safe"
s1=s.swapcase()
print(s)
print(s1)
s2=s.title() # makes the first letter of each word capital
print(s2)
s3=s.capitalize() # makes only the first letter capital
print(s3)
print("python".capitalize())
```

Output:

```
In [45]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
STAY HOME stay safe
stay home STAY SAFE
Stay Home Stay Safe
Stay home stay safe
Python
```

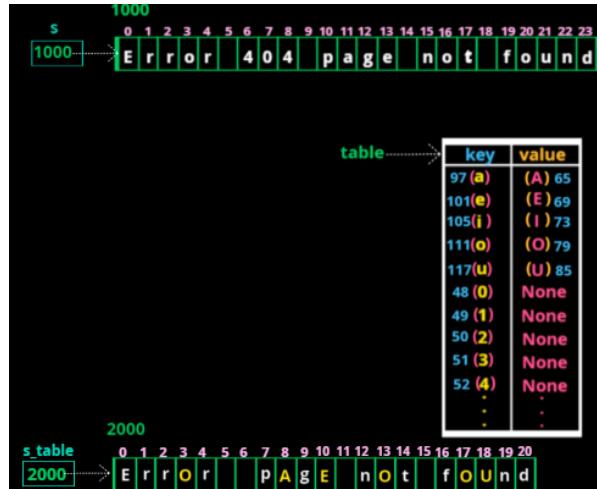


String translation

Python string method **translate()** returns a copy of the string in which all characters have been **translated** using table (constructed with the **maketrans()** function in the **string** module)

```
s="Error 404 not found"
table=s.maketrans("aeiou","AEIOU","0123456789")
s_table=s.translate(table)
print(s)
print(s_table)
```

Logic:



Output:

```
In [46]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
Error 404 not found
ErrOr nOt fOUnd
```

Python Fundamentals

day 20

Today's Agenda

- String formatting
- Format specification types
- Programs



String formatting

In python we have two ways to achieve string formatting

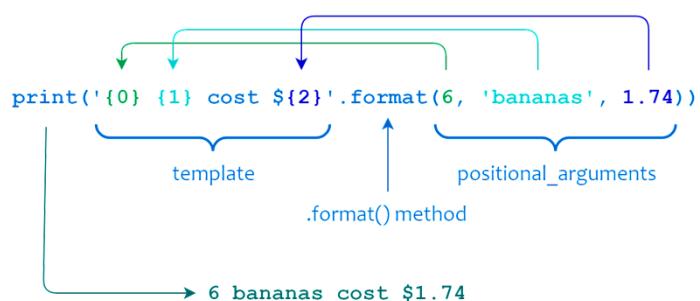
- ❖ `format()` method
- ❖ f string literal

`format()` method

The `format()` method formats the specified value(s) and insert them inside the string's placeholder. The placeholder is defined using curly brackets `{ }`.

Syntax:

string.format(*args)



```
name=input("Enter your name\n")
place=input("Enter your place\n")

s="Hello {}, How is the weather in {}?".format(name,place)
print(s)
```

Output:

```
In [1]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
```

```
Enter your name
rohit
```

```
Enter your place
blore
Hello rohit, How is the weather in blore?
```



Let us see another example

```
s="{} {} {}".format(10,20,30)
print(s)
```

Output:

```
In [2]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
10 20 30
```

Note: If nothing is specified in place holders, the default order is considered.

Now let us see an example where order is specified explicitly

```
s="{2} {0} {1}".format(10,20,30)
print(s)
```

Output:

```
In [3]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
30 10 20
```

Within the place holder { } not only can we pass positional values, but also additional values which is going to change the way of its representation in different ways.

string → "position:format_specification"

format specification types:

- Alignment
- Presentation
- Conversion



Format specification

Alignment:

- >right alignment

```
s="{0:*>10}".format(999) #right alignment with * as the fill  
print(s)
```

Output:

```
In [5]: runfile('C:/Users/rooman/OneDrive/  
Desktop/python/test.py', wdir='C:/Users/rooman/  
OneDrive/Desktop/python')  
*****999
```

- <left alignment

```
s="{0:*<10}".format(999) #Left alignment with * as the fill  
print(s)
```

Output:

```
In [4]: runfile('C:/Users/rooman/OneDrive/  
Desktop/python/test.py', wdir='C:/Users/rooman/  
OneDrive/Desktop/python')  
999*****
```

- ^ center alignment

```
s="{0:^^11}".format(999) #center alignment with * as the fill  
print(s)
```

Output

```
In [6]: runfile('C:/Users/rooman/OneDrive/  
Desktop/python/test.py', wdir='C:/Users/rooman/  
OneDrive/Desktop/python')  
****999****
```

Presentation:

- f-fixed point notation

```
import math
s="{0:.4f}".format(math.pi) #print pi value with 4 decimal points
print(s)
```

Output:

```
In [7]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
3.1416
```

We can also have pi value beginning with 0's and mentioning number of digits as follows

```
import math
s="{0:010.4f}".format(math.pi)
print(s)
```

Output:

```
In [8]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
00003.1416
```

- e-exponent notation

```
ew=5976000000000000000000000000
s="{0:e}".format(ew)
print(s)
```

Output:

```
In [9]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
5.976000e+23
```

As we know we can adjust the decimal numbers let us see how we can do it here

```
ew=5976000000000000000000000000
s="{0:.3e}".format(ew)
print(s)
```

Output:

```
In [10]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
5.976e+23
```

Let us now try to pass different arguments inside place holders.

```
s="{} {} {}".format([10,20,30]) #passing list as input
print(s)
```

Output:

```
File "C:\Users\rooman\Anaconda3\lib\site-
packages\spyder_kernels\customize
\spydercustomize.py", line 110, in execfile
    exec(compile(f.read(), filename, 'exec'),
namespace)

File "C:/Users/rooman/OneDrive/Desktop/
python/test.py", line 1, in <module>
    s="{} {} {}".format([10,20,30]) #passing
list as input

IndexError: tuple index out of range
```



The above input will give an error because we have three place holders but we are trying to pass three inputs in a single list. Is there a way to tackle this? Definitely, let us see how

```
s="{0[0]} {0[1]} {0[2]}".format([10,20,30]) #passing list as input
print(s)
```

Output:

```
In [12]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
10 20 30
```

However there's also an easy way to do this called as **unpacking** as shown below

```
s="{0} {1} {2}".format(*[10,20,30]) #unpack by using *
print(s)
```

Output:

```
In [13]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
10 20 30
```



Programs

- Let us take numbers as input from user and try to find the average of numbers with only 4 decimal points.

```
nums=input("Enter the number\n").split() #splits the number present in string
print(nums)
print(type(nums))
```

Output:

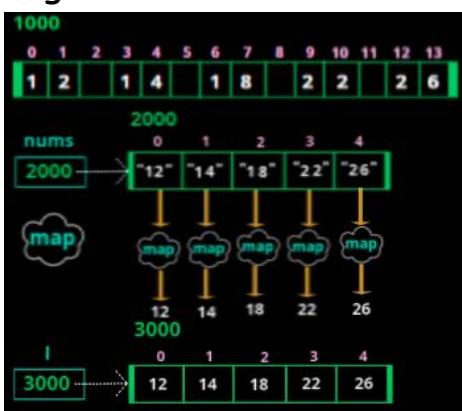
```
In [14]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
```

```
Enter the number
12 14 18 22 26
['12', '14', '18', '22', '26']
<class 'list'>
```

We now have the list of numbers, but we cannot perform any math operations on list of strings. So now we have to map strings to integers as shown below

```
nums=input("Enter the number\n").split() #splits the number present in string
print(nums)
l=list(map(int,nums))
print(l)
```

Logic:



Output:

```
In [16]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
```

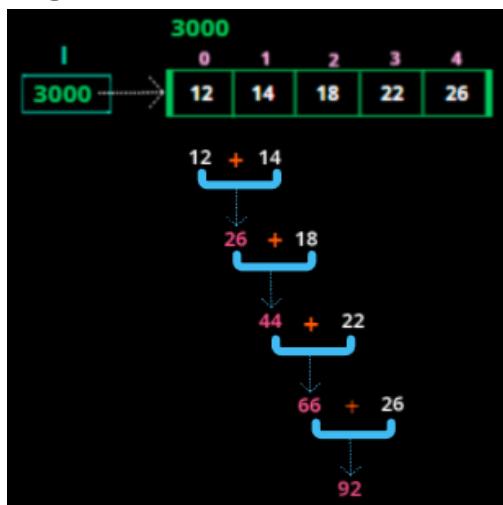
```
Enter the number
12 14 18 22 26
['12', '14', '18', '22', '26']
[12, 14, 18, 22, 26]
```

Now we have list of integers, next we should reduce the list to the sum of all the integers by using `reduce()` function

```
from functools import reduce

nums=input("Enter the number\n").split() #splits the number present in string
l=list(map(int,nums))
res=reduce(lambda x,y:x+y,1)
print(res)
```

Logic:



Output:

```
In [17]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
```

```
Enter the number
12 14 18 22 26
92
```

As expected we got the sum of all the integers, now we should take the average by dividing it with number of integers present as shown below

```
from functools import reduce

nums=input("Enter the number\n").split() #splits the number present in string
l=list(map(int,nums))
res=reduce(lambda x,y:x+y,1)
avg=res/len(l)
print("{0:.4f}".format(avg))
print("{0:^14.4f}".format(avg)) #center align to 14
```

Output:

```
In [18]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
```

```
Enter the number  
12 14 18 22 26  
18.4000  
18.4000
```



Try to solve such programs which involve application of the functions you have studied so far and join the dots.



Python Fundamentals

day 21

Today's Agenda

- String replication
- String formatting-Conversion
- 'f' string literal
- Raw string literal
- Regular expressions



String Replication

We know what is string concatenation, let us know what is string replication with the help of following example where we are trying to replicate the string **python** three times

```
s="PYTHON"  
s_rep=s*3 # * is the replication operator in python  
print(s)  
print(s_rep)
```

Output:

```
In [19]: runfile('C:/Users/rooman/OneDrive/  
Desktop/python/test.py', wdir='C:/Users/rooman/  
OneDrive/Desktop/python')  
PYTHON  
PYTHONPYTHONPYTHON
```

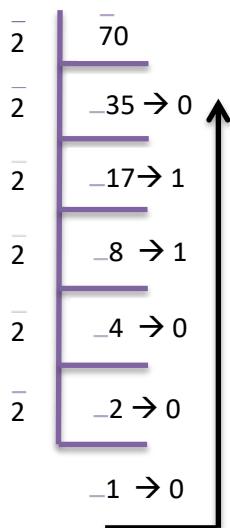
We can repeat the single string value the amount of times equivalent to the integer value passed.

String formatting- Conversion

- b- decimal to binary

```
a=70  
print(a)  
print("{0:b}".format(a)) #converting decimal to binary
```

Logic:



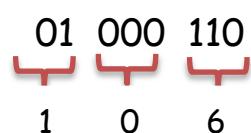
Output:

```
In [20]: runfile('C:/Users/rooman/OneDrive/  
Desktop/python/test.py', wdir='C:/Users/rooman/  
OneDrive/Desktop/python')  
70  
1000110
```

- o- decimal to octal

```
a=70  
print(a)  
print("{0:o}".format(a)) #converting decimal to octal
```

Logic:



Output:

```
In [21]: runfile('C:/Users/rooman/OneDrive/  
Desktop/python/test.py', wdir='C:/Users/rooman/  
OneDrive/Desktop/python')  
70  
106
```

- x- decimal to hexadecimal

```
a=70
print(a)
print("{0:x}".format(a)) #converting decimal to hexadecimal
```

Logic:

0100	0110
4	6



Output:

```
In [22]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
70
46
```

‘f’ string literal

As of Python 3.7, f-strings are a great new way to format strings. Not only are they more readable, more concise, and less prone to error than other ways of formatting, but they are also faster!

Let us understand by an example

format()

```
import math
name='rohit'
place='Blore'
print("{} {}".format(name,place))
print("{1} {0}".format(name,place))
print("{0:.4f}".format(math.pi))
```

Output:

```
In [25]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
rohit Blore
Blore rohit
3.1416
```

f-string

```
import math
name='rohit'
place='Blore'
print(f'{name} {place}')
print(f'{place} {name}')
print(f'{math.pi:.4f}')
```

In [27]:

```
In [27]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
rohit Blore
Blore rohit
3.1416
```

Let us compare the performance of both as below

```
import timeit
print(timeit.timeit(stmt="{0:.2f}".format(3.1416), number=1000))
print(timeit.timeit(stmt=f"{3.1416:.2f}", number=1000))
```

Output:

```
In [35]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
9.599985787644982e-06
3.100000321865082e-05
```

As we can see **f-string** is much better than **format()** from above output. The timing changes with respect to the processor.

Raw string literal

Let us see an example to understand the meaning of raw string

```
name="Ro\nhit" #normal string
name=r"Ro\nhit" #raw string
print(name)
```

Output:

```
In [36]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
Ro
hit
```

```
In [37]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
Ro\nhit
```



Regular Expressions

Regular expression is a sequence of characters that forms a search pattern. Why do we need it? Where it is used? Let us see

- In **MS word** whenever there is a spelling mistake, we get red underline which notifies us that the spelling is wrong. This is done by matching the word we enter to the actual word and if there is a mismatch then it notifies by the red line.
- **Google search engine**, where anything that we entered is matched with whatever is there in google. It does this with the help of software called as webscroller who scans the entire internet to find the search queries matching the entered data.
- **Spam filtering** is another best example where all our mails are being monitored and filtered based on the words present in the content of it. Every mail goes through the spam filter where if words like **free, discount, cheap, offer etc** are encountered then it is a spam.



Let us see an example and understand in a better way

```
import re

text="Python is easy"
regex=r"Python" # regular exp should always be a raw string,
match=re.match(regex,text)
print(match)
```

Output:

```
In [39]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
<re.Match object; span=(0, 6), match='Python'>
```

The output says that match is found in the span of 0 to 6 that is Python. Let us see how to access that in a better way

```
import re

text="Python is easy"
regex=r"Python" # regular exp should always be a raw string,
match=re.match(regex,text)
start,end=match.span()
print(text[match.start():match.end():])
```

Output:

```
In [41]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
Python
```

Let us now try to find another match



```
import re

text="Python is super easy"
regex=r"super" # regular exp should always be a raw string.
match=re.match(regex,text)
```

Output:

```
In [45]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
None
```

We can see that we did not get any match object. Which means the **match()** starts searching from the beginning and if it doesn't find the match in starting then it doesn't return any match object.

Is there a way to overcome this disadvantage? Definitely there is, let us see how

```
import re

text="Python is super easy"
regex=r"super" # regular exp should always be a raw string
print(re.search(regex,text))
```

Output:

```
In [46]: runfile('C:/Users/rooman/OneDrive/  
Desktop/python/test.py', wdir='C:/Users/rooman/  
OneDrive/Desktop/python')  
<re.Match object; span=(10, 15), match='super'>
```

By making use of `search()`, irrespective of the position the matching is done. So we have found a match object in span of 10 to 15 where 15 is exclusive.

```
import re  
  
text="Python is super easy"  
regex=r"super" # regular exp should always be a raw string  
match=re.search(regex,text)  
print(text[match.start():match.end():])
```

Output:

```
In [47]: runfile('C:/Users/rooman/OneDrive/  
Desktop/python/test.py', wdir='C:/Users/rooman/  
OneDrive/Desktop/python')  
super
```

This is how we slice the match object.

From the above examples we understood that `match()` can be used for pattern matching where the match object is present in the beginning whereas `search()` will do the same irrespective of the position of the match object.



Python fundamentals

day 22

Today's Agenda

- Meta characters
- Programs



Meta characters

A character when used in regular expression has a special meaning to it is called meta character. There are many meta characters but now let's focus on **.**(dot) which will match single character.

Let us understand with examples

```
import re
text="Python is super easy"
regex=r"." #meta character
match=re.search(regex,text)
print(match)
```

Output:

```
In [51]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
<re.Match object; span=(0, 1), match='P'>
```

search() will only give the first match. If we want to see all the matches then we should use another function called as **findall()**. Let us see how to use it

```
import re
text="Python is super easy"
regex=r"." #meta character
l=re.findall(regex,text)
print(l)
```

Output:

```
In [52]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
[ 'P', 'y', 't', 'h', 'o', 'n', ' ', 'i', 's', ' ', 's', 'u', 'p', 'e', 'r', ' ', 'e', 'a', 's', 'y' ]
```



We can see that all the matches have been found and a list of every single character is created including spaces and full point.

Now what if we want to match the .(full point) in the test string, let us see how to do it

```
import re
text="Python is super easy."
regex=r"\." #escape meta character
l=re.findall(regex,text)
print(l)
```

Output:

```
In [54]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
[ '.' ]
```

**** will escape the meaning of **.** as meta character. That is why \ is called escape(special character).

Now what if we want to find multiple matches, let us see how to find them

```
import re
text="Python is super easy."
regex=r"Python|super" #or meta character
l=re.findall(regex,text)
print(l)
```

Output:

```
In [55]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
['Python', 'super']
```

|(pipe) will search for Python or super. Sometimes only one amongst them can be present and sometimes both, either way | meta character works.

Now let us consider another example

```
import re
text="a whole hole is not a wwwhole"
regex=r"whole"
l=re.findall(regex,text)
print(l)
```

Output:

```
In [56]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
['whole', 'whole']
```



We can see only two matches are found, what if we want the match where w is missing or in other words 0 or many occurrence should be considered. Let us see which meta character to use then

```
import re
text="a whole hole is not a wwwhole"
regex=r"w*hole" # * is a 0 or many meta character
l=re.findall(regex,text)
print(l)
```

Output:

```
In [57]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
['whole', 'hole', 'wwwwhole']
```

For 0 or many * is the meta character to be used. Similarly if we want only 0 or 1 occurrence then ? should be used as below

```
import re
text="a whole hole is not a wwwhole"
regex=r"w?hole" # ? is a 0 or 1 meta character
l=re.findall(regex,text)
print(l)
```

Output:

```
In [58]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
['whole', 'hole', 'whole']
```

Now let us see how to search for 1 or many

```
import re
text="a whole hole is not a wwwhole"
regex=r"w+hole" # + is a 1 or many meta character
l=re.findall(regex,text)
print(l)
```

Output:

```
In [59]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
['whole', 'wwwhole']
```



Let us see another example to understand better

```
import re
text="I know that no one is there in the school now"
regex=r"k?now?" # ? is a 0 or 1 meta character
l=re.findall(regex,text)
print(l)
print("Number of occurrences are ",len(l))
```

Output:

```
In [61]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
['know', 'no', 'now']
Number of occurrences are 3
```

Now let us see another meta character where we can search for exact number of occurrences

```
import re
text='''
google
google
goooogle
gooooole
oooooooole
oooooooooole'''
regex=r"go{2,5}gle" # inside {} range can be specified
l=re.findall(regex,text)
print(l)
print("Number of occurrences are ",len(l))
```



Output:

```
In [62]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
['google', 'google', 'goooogle', 'gooooole']
Number of occurrences are 4
```

Let us see another example where multiple occurrences of python is there, but we want the match that is at beginning.

```
import re
text="python has nothing to do with snake python"
regex=r"^\w+python"
match=re.search(regex,text)
print(match)
```

Output:

```
In [63]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
<re.Match object; span=(0, 6), match='python'>
```

^ helps to search for the particular pattern in beginning of the string. We see that a match object is found in the span of 0 to 6 which is python.

Similarly to find the match at the end we have another meta character \$,

```
import re
text="python has nothing to do with snake python"
regex=r"python$"
match=re.search(regex,text)
print(match)
```

Output:

```
In [65]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
<re.Match object; span=(36, 42),
match='python'>
```

Now let us take another example where we want to match all the vowels present in the test string.

```
import re
text="python java ai data science"
regex=r"[aeiou]" # [] meta character
l=re.findall(regex,text)
print(l)
print("Number of occurrences=",len(l))
```

Output:

```
In [66]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
['o', 'a', 'a', 'a', 'i', 'a', 'a', 'i', 'e',
'e']
Number of occurrences= 10
```



Character class meta character [] searches for any of the mentioned characters in the test string.

Now what if we want to search all the characters which are not vowels/consonants, let us see what to do

```
import re
text="python java ai data science"
regex=r"[^aeiou]"
l=re.findall(regex,text)
print(l)
print("Number of occurrences=",len(l))
```

Output:

```
In [67]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
['p', 'y', 't', 'h', 'n', ' ', 'j', 'v', ' ', ' '
, 'd', 't', ' ', 's', 'c', 'n', 'c']
Number of occurrences= 17
```

Note: ^ used inside [] will match all the characters except the ones specified in [].

Now let us try to match all the lower case, upper case and the digits in the test string using character class

```
import re
text="My name is ROHIT and this is my number: 906612345"
regex=r"[a-zA-Z0-9]"
l=re.findall(regex,text)
print(l)
print("Number of occurrences=",len(l))
```

Output:

```
In [68]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
['M', 'y', 'n', 'a', 'm', 'e', 'i', 's', 'R',
'O', 'H', 'I', 'T', 'a', 'n', 'd', 't', 'h',
'i', 's', 'i', 's', 'm', 'y', 'n', 'u', 'm',
'b', 'e', 'r', '9', '0', '6', '6', '1', '2',
'3', '4', '5']
Number of occurrences= 39
```



As we can see all the lower case has been matched by saying a-z, upper case by saying A-Z and digits by saying 0-9.

The above can be performed in other way as well, by using another meta character \w which will match all alpha numeric characters

```
import re
text="My name is ROHIT and this is my number: 906612345"
regex=r"\w"
l=re.findall(regex,text)
print(l)
print("Number of occurrences=",len(l))
```

Output:

```
In [69]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
['M', 'y', 'n', 'a', 'm', 'e', 'i', 's', 'R',
'O', 'H', 'I', 'T', 'a', 'n', 'd', 't', 'h',
'i', 's', 'i', 's', 'm', 'y', 'n', 'u', 'm',
'b', 'e', 'r', '9', '0', '6', '6', '1', '2',
'3', '4', '5']
Number of occurrences= 39
```

Now let us match only the special characters

```
import re
text="My name is ROHIT and this is my number: 90661:
regex=r"[^a-zA-Z0-9]"
l=re.findall(regex,text)
print(l)
print("Number of occurrences=",len(l))
```

Output:

```
In [70]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
[', ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ':', ' '
]
Number of occurrences= 10
```



Is there an easier way to do this? Definitely, by using \W

```
import re
text="My name is ROHIT and this is my number: 906612345"
regex=r"\W" # \W is not alpha numeric
l=re.findall(regex,text)
print(l)
print("Number of occurrences=",len(l))
```

Output:

```
In [71]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
[ ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ':', ' '
']
Number of occurrences= 10
```

\w is for alpha numeric and \W is for non-alpha numeric characters.

Similarly there are shortcuts to find only digits by using \d, and non-digits by using \D.

```
import re
text="My name is ROHIT and this is my number: 906612345"
regex=r"\d" # only digits
l=re.findall(regex,text)
print(l)
print("Number of occurrences=",len(l))
```

Output:

```
In [72]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
['9', '0', '6', '6', '1', '2', '3', '4', '5']
Number of occurrences= 9
```



Now let's see for non-digits

```
import re
text="My name is ROHIT and this is my number: 906612345"
regex=r"\D" # non-digits
l=re.findall(regex,text)
print(l)
print("Number of occurrences=",len(l))
```

Output:

```
In [73]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
['M', 'y', ' ', 'n', 'a', 'm', 'e', ' ', 'i',
's', ' ', 'R', 'O', 'H', 'I', 'T', ' ', 'a',
'n', 'd', ' ', 't', 'h', 'i', 's', ' ', 'i',
's', ' ', 'm', 'y', ' ', 'n', 'u', 'm', 'b',
'e', 'r', ':', ' ']
Number of occurrences= 40
```

Similarly, if you want only white spaces to be matched you can use \s and try for yourself.

To summarize the list of meta characters studied so far

- . → single character
- \ → escape (special character)
- | → A|B or operator
- * → 0 or many
- ? → 0 or 1
- + → 1 or many
- {m} → matches exactly m characters
- {m,n} → matches between m and n characters
- [] → character class
- ^ → start of string
- \$ → end of string
- \s → any white space character
- \S → any non-white space character
- \d → any digit
- \D → any non-digit
- \w → any word character
- \W → any non-word character
- \b → a word boundary
- \B → non-word boundary



Programs

Now let's see how to combine the meta characters based on our expectation

Example 1: To find the matches of vowels occurring twice

```
import re
text="If foot becomes feet tooth becomes teeth"
regex=r"[aeiou]{2}" #will match for vowels that has occurred twice
l=re.findall(regex,text)
print(l)
print("Number of occurrences=",len(l))
```

Output:

```
In [76]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
['oo', 'ee', 'oo', 'ee']
Number of occurrences= 4
```

Example 2: To find the matches week and weak

```
import re
text="Only the weak wait for the week to end"
regex=r"we[ae]k" # [ae] will match for either e or a
l=re.findall(regex,text)
print(l)
print("Number of occurrences=",len(l))
```

Output:

```
In [78]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
['weak', 'week']
Number of occurrences= 2
```



Python fundamentals

day 23

Today's Agenda

- Word boundary
- Programs on meta characters
- Grouping
- Programs



Word boundary

Word boundary is another meta character which will put boundaries wherever it is used. From the below example this can be understood in a better way

```
import re
text='''abcpqrxyz
pqrxyzabc
pqrabbcxyz
abc'''
regex=r"\babc" #will match the string where nothing is there before abc
l=re.findall(regex,text)
print(l)
```

Output:

```
In [1]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
['abc', 'abc']
```

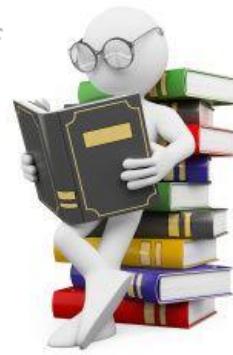
Here **abcpqrxyz** and **abc** got matched as there is nothing before abc.

\b will put a word boundary, where if **\b** is at the beginning of regex then if anything present before the boundary it doesn't get matched. And if **\b** is at the end of regex then if anything is present after the boundary it doesn't get matched.

```
import re
text='''abcpqrxyz
pqxyzabc
pqrabcxyz
abc'''
regex=r"abc\b" #will match the string where nothing is there after abc
l=re.findall(regex,text)
print(l)
```

Output:

```
In [2]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
['abc', 'abc']
```



Here **pqxyzabc** and **abc** got matched as there is nothing after abc in these strings.

```
import re
text='''abcpqrxyz
pqxyzabc
pqrabcxyz
abc'''
regex=r"\babc\b" #will match the string where nothing is there before/after abc
l=re.findall(regex,text)
print(l)
```

Output:

```
In [3]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
['abc']
```

We saw that in all the above three cases **pqrabcxyz** didn't get matched. So to get a match of string which has characters on either side, we should use **\B** which is opposite of word boundary.

```
import re
text='''abcpqrxyz
pqxyzabc
pqrabcxyz
abc'''
regex=r"\Babc\B" #will match the string where either side characters are present
l=re.findall(regex,text)
print(l)
```

Output:

```
In [4]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
['abc']
```

To cross verify, let us print it using **search()** to get the span of matched object.

```
import re
text='''abcpqrxyz
pqxyzabc
pqrabcxyz
abc'''
regex=r"\Babc\B" #will match the string where either side characters are present
match=re.search(regex,text)
print(match)
```

Output:

```
In [5]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
<re.Match object; span=(23, 26), match='abc'>
```

Programs on meta characters

Example 1: To match first word from the sentence.

```
import re
text="Python is the best language"
regex=r"^[a-zA-Z]+"
match=re.search(regex,text)
print(match)
print(match.group())
```



Output:

```
In [6]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
<re.Match object; span=(0, 6), match='Python'>
Python
```

Example 2: To match last word from the sentence.

```
import re
text="Python is the best language"
regex=r"[a-zA-Z]+$"
match=re.search(regex,text)
print(match)
print(match.group())
```

Output:

```
In [7]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
<re.Match object; span=(19, 27),
match='language'>
language
```

Example 3: Let's match the words in the test string having the word length as 4.

```
import re
text="Python is the best language"
regex=r"\b[a-zA-Z]{4}\b"
l=re.findall(regex,text)
print(l)
```

Output:

```
In [9]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
['best']
```



Example 4: To check if the given email address is valid address or not. With the following rule set

- Username:
 - a. Character: alphanumeric
 - b. Special character: _, - or \$
- Domain: gmail.com

```
import re
text='''rohit@gmail.com
rohit@gmai.com
rohit_xyg@gmail.com
roh?>@gmail.com
rohit-123@gmail.com'''
regex=r"[a-zA-Z0-9_\-]+@[a-zA-Z]+\.[a-zA-Z]+"
l=re.findall(regex,text)
print(l)
```

Output:

```
In [10]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
['rohit@gmail.com', 'rohit_xyg@gmail.com',
'rohit-123@gmail.com']
```



Example 5: Similar to previous example but here let us consider other domains as well

```
import re
text='''rohit@gmail.com
rohit@gmai.com
rohit_xyg@gmail.com
roh?>@gmail.com
rohit-123@gmail.com
rohit@yahoo.com
rohit@outlook.com
rohit@hotmail.com'''
regex=r"[a-zA-Z0-9_\-]+@[a-zA-Z]+\.[a-zA-Z]+"
l=re.findall(regex,text)
print(l)
```

Output:

```
In [11]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
['rohit@gmail.com', 'rohit_xyg@gmail.com',
'rohit-123@gmail.com', 'rohit@yahoo.com',
'rohit@outlook.com', 'rohit@hotmail.com']
```

Grouping

Whatever a regular expression matches for, that complete string is actually treated as group 0. Let us verify this

```
import re
text='''rohit@gmail.com
rohit@gmail.com
rohit_xyz@gmail.com
roh?>@gmail.com
rohit-123@gmail.com
rohit@yahoo.com
rohit@outlook.com
rohit@hotmail.com'''
regex=r"[a-zA-Z0-9_\-]+@[a-zA-Z]+\.[com]"
match=re.search(regex,text)
print(match.group(0))
```

Output:

```
In [12]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
rohit@gmail.com
```



Now we know that match object belongs to **group 0**, as we used **search()** we could match only first occurrence.

What if we want to create **subgroups** within regular expression? Is that possible? Definitely it is possible by using **()**, let us see how

```
import re
text='''rohit@gmail.com
rohit@gmail.com
rohit_xyz@gmail.com
roh?>@gmail.com
rohit-123@gmail.com
rohit@yahoo.com
rohit@outlook.com
rohit@hotmail.com'''
regex=r"([a-zA-Z0-9_\-]+)@([a-zA-Z]+\.[com])" #subgrouping username and domain
match=re.search(regex,text)
print(match.group(0)) #will give entire string
print(match.group(1)) #will give username
print(match.group(2)) #will give domain
```

Output:

```
In [14]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
rohit@gmail.com
rohit
gmail.com
```

Great!!! But we could only match first occurrence, what about the rest? How should we match them? Let us see if there is a solution for it

```
import re
text='''
rohit@gmail.com
rohit@gmail.com
rohit_xyz@gmail.com
roh?>@gmail.com
rohit-123@gmail.com
rohit@yahoo.com
rohit@outlook.com
rohit@hotmail.com'''
regex=r"([a-zA-Z0-9_\-]+)@([a-zA-Z]+\.[com])" #subgrouping username and domain
itr=re.finditer(regex,text)

for m in itr:
    print(m.group(0))
    print(m.group(1))
    print(m.group(2))
```

Output:

```
In [15]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
rohit@gmail.com
rohit
gmail.com
rohit_xyz@gmail.com
rohit_xyz
gmail.com
rohit-123@gmail.com
rohit-123
gmail.com
rohit@yahoo.com
rohit
yahoo.com
rohit@outlook.com
rohit
outlook.com
rohit@hotmail.com
rohit
hotmail.com
```



Programs

Example 1: From the previous code, replace the domain name to "@rooman.com"

```
import re
text='''rohit@gmail.com
rohit@gmail.com
rohit_xyz@gmail.com
roh?>@gmail.com
rohit-123@gmail.com
rohit@yahoo.com
rohit@outlook.com
rohit@hotmail.com'''
regex=r"@[a-zA-Z]+\..com"
s=re.sub(regex,"@rooman.com",text)
print(s)
```

Output:

```
In [19]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
rohit@rooman.com
rohit@@rooman.com
rohit_xyz@rooman.com
roh?>@rooman.com
rohit-123@rooman.com
rohit@rooman.com
rohit@rooman.com
rohit@rooman.com
```



Example 2: Find the number of times substitution has occurred in previous example.

```
import re
text='''rohit@gmail.com
rohit@gmail.com
rohit_xyz@gmail.com
roh?>@gmail.com
rohit-123@gmail.com
rohit@yahoo.com
rohit@outlook.com
rohit@hotmail.com'''
regex=r"@[a-zA-Z]+\..com"
s=re.subn(regex,"@rooman.com",text)
print(s)
```

Output:

```
In [20]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
('rohit@rooman.com\nrohit@@rooman.com
\nrohit_xyz@rooman.com\nroh?>@rooman.com
\nrohit-123@rooman.com\nrohit@rooman.com
\nrohit@rooman.com\nrohit@rooman.com', 8)
```

From above output, we can see that substitutions have happened 8 times.

Example 3: Given the text "1992/04/10", split the year, month and day using re module.

```
import re
text="1992/04/10"
l=re.split(r"[/-]",text) #do not place - after / otherwise
print(l)                 #it will act like escape character
```

Output:

```
In [22]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
['1992', '04', '10']
```



Python Fundamentals

day 24

Today's Agenda

- Pattern object
- Data structures
- Disadvantages of array



Pattern object

An object which has regular expression inside is called pattern object.

Approach 1

```
import re
text="9912345689 99853271"
regex=r"\d{10}"
print(re.search(regex,text))
print(re.findall(regex,text))
```

Output:

```
In [12]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
<re.Match object; span=(0, 10),
match='9912345689'>
['9912345689']
```

Approach 2

```
import re
text="9912345689 99853271"
p=re.compile(r"\d{10}")
print(type(p))
print(p.search(text))
print(p.findall(text))
```

Output:

```
In [13]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
<class 're.Pattern'>
<re.Match object; span=(0, 10),
match='9912345689'>
['9912345689']
```

No approach is better than the other it's the user comfort and their preference.

Data structures

Python has four main data structures which are shown below



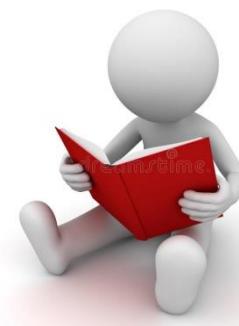
These are called data structures because each of these store data in different manner and access the data in different manner along with some restrictions of each due to which their behaviour and application differ.

Let us understand how and where to use data structures exactly.

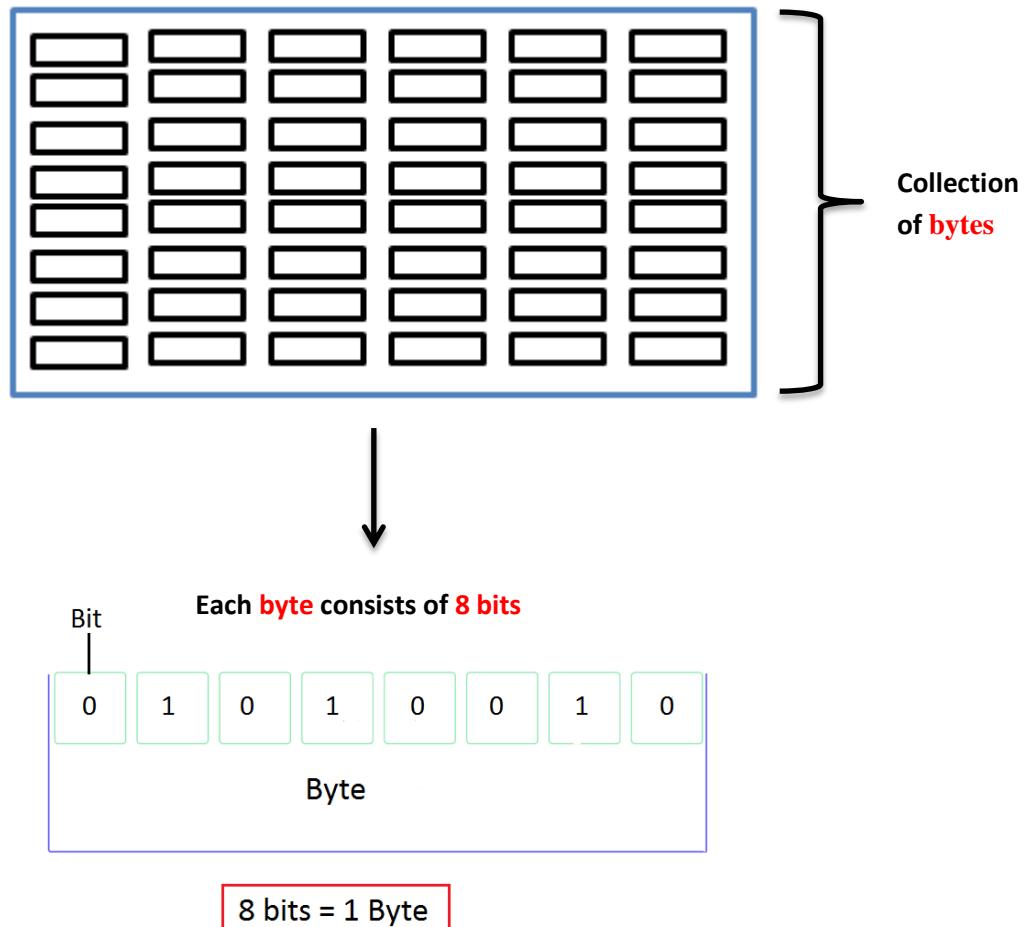


Before learning about data structures, one should always know how a data is stored in RAM. RAM is a collection of bytes, byte consists of the fundamental building block called bits. 8 bits is one byte. RAM is collection of crores of such bytes. Data in these bytes is stored in machine language(binary) which is combination of 1's and 0's.

This is how **RAM** looks like

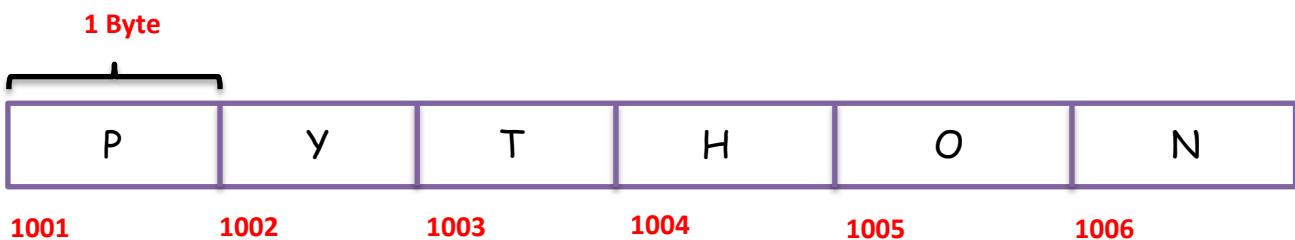


Internal representation of **RAM**



Inside each byte data is stored as **combination of 1's and 0's**

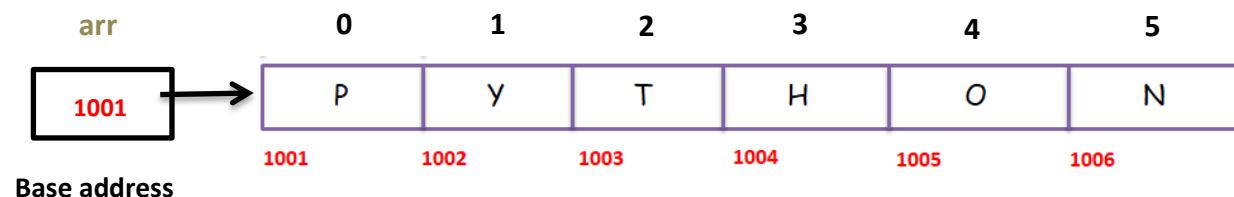
Let us now store PYTHON in RAM and see how exactly it is stored



In each byte a character gets stored but in binary format, for your understanding purpose it is shown in characters.

Wherever you see continuous storage of characters or one next to other it is referred to as array in any programming language. Technically continuous storage is called as contiguous memory allocation.

In such cases the address of first byte is stored in the reference which is called as the base address, which will point to the array.



If data is stored contiguously like this and if **arr** has the base address then you can access any element present in this array as efficiently and soon as possible. Wonder how? The answer is using the formula called **array arithmetic**.

arr[index] = base address + width * index

$$\text{arr}[3] = 1001 + 1 * 3$$

$$\text{arr}[3] = 1004$$

Arrays also come with some disadvantages.

Let us see what are those

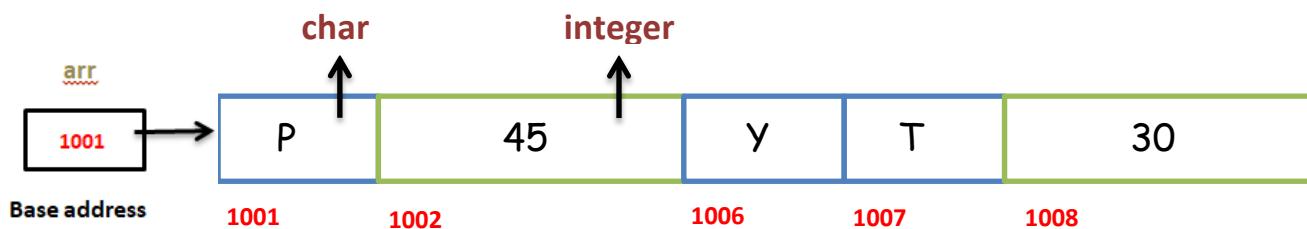


Disadvantages of arrays

- ❖ Arrays require contiguous memory allocation.
- ❖ Arrays can store only homogeneous data.
- ❖ The size of an array can neither grow nor shrink.

Let us see the first two drawbacks or disadvantages.

We know that different type of data occupy different size in memory, let us consider the following example to understand in a better way



Now let us try to apply array arithmetic formula and see if it will work

`arr[3] = 1001 + 1 * 3 //there no uniform width of data`

`arr[3] = 1004 //which is not the right address`

We see that there is mismatch in addresses. This means, if we try to store different types of data, arrays will fail because arrays arithmetic doesn't stand correct for this situation.



Python fundamentals

day25

Today's Agenda

- Disadvantages of arrays(contd)
- Referential array
- Lists
- Operations on list

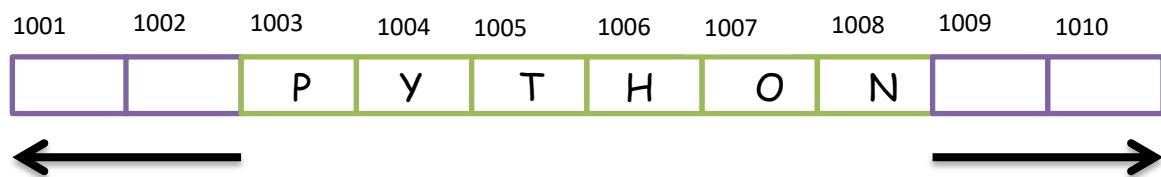


Disadvantages of arrays (contd)

In previous session we saw two disadvantages, let us understand the third disadvantage of arrays

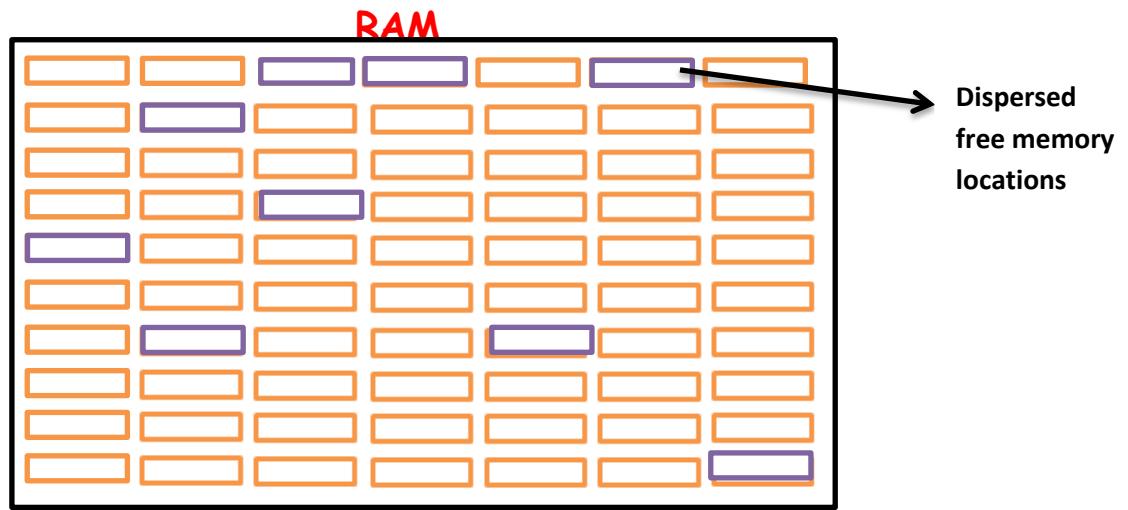
The third disadvantage is that the **array can neither grow nor shrink in size**. Let us try and understand why this is so

Consider the following situation where **PYTHON** is stored in certain memory



There is a possibility that after memory for **PYTHON** was allocated the memory beside **PYTHON** was assigned for some other operation.

This means that the array cannot grow contiguously in either direction.



The other way of understanding this is, free memory locations are very rarely found next to each other. Most of the time they are dispersed in RAM, and we know arrays definitely cannot make use of dispersed memory locations as it won't satisfy arrays arithmetic formula. But there are other data structures like linked lists, which can link these dispersed memory locations and form another data structure.

Referential array

Python has a different take on arrays because it uses referential arrays, let us see what this means

Consider the following example where different type data is present

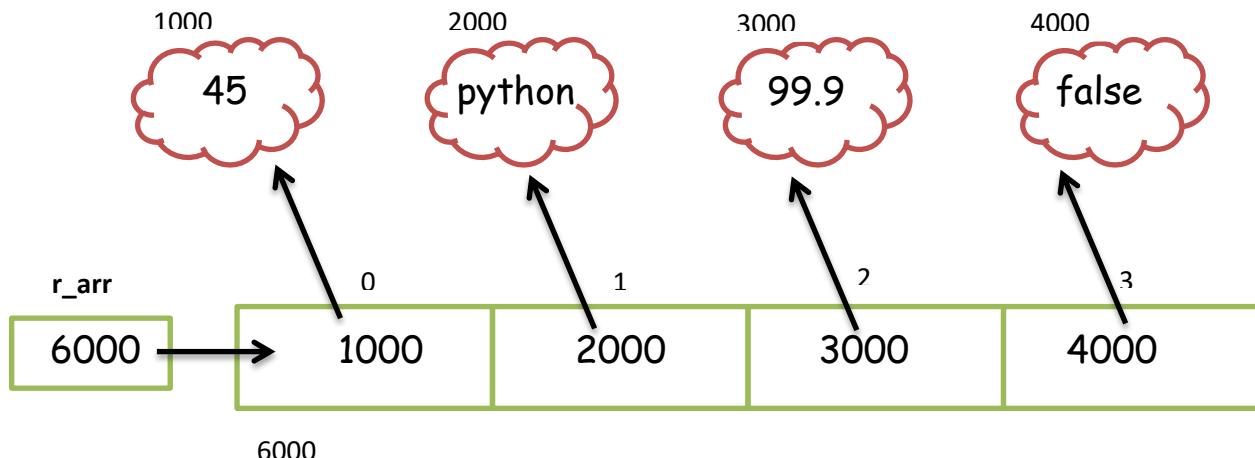
45 → integer

Python → String

99.9 → float

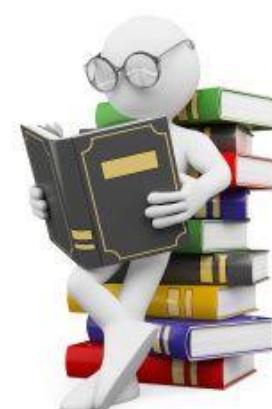
False → Boolean

Motive is to store these data contiguously, let us see how



The solution to store different types of data in a single array is by storing the addresses of the objects rather than their values, the addresses will refer/point the objects and hence this type of array is called referential arrays. As the addresses will be of same type there won't be a problem in giving base address as the reference of the array which will point to it and satisfy array arithmetic formula as well.

Most of the data structures in python make use of referential arrays. Strings are also stored in the same format, where each character's object is created with its own address and these addresses are stored in an array.



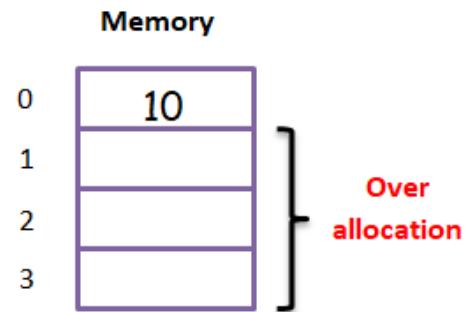
Lists

Let us start with the first data structure list, which uses dynamic array. Dynamic array is an array which can grow and shrink in size. Confused? Let us understand the explanation shall we?

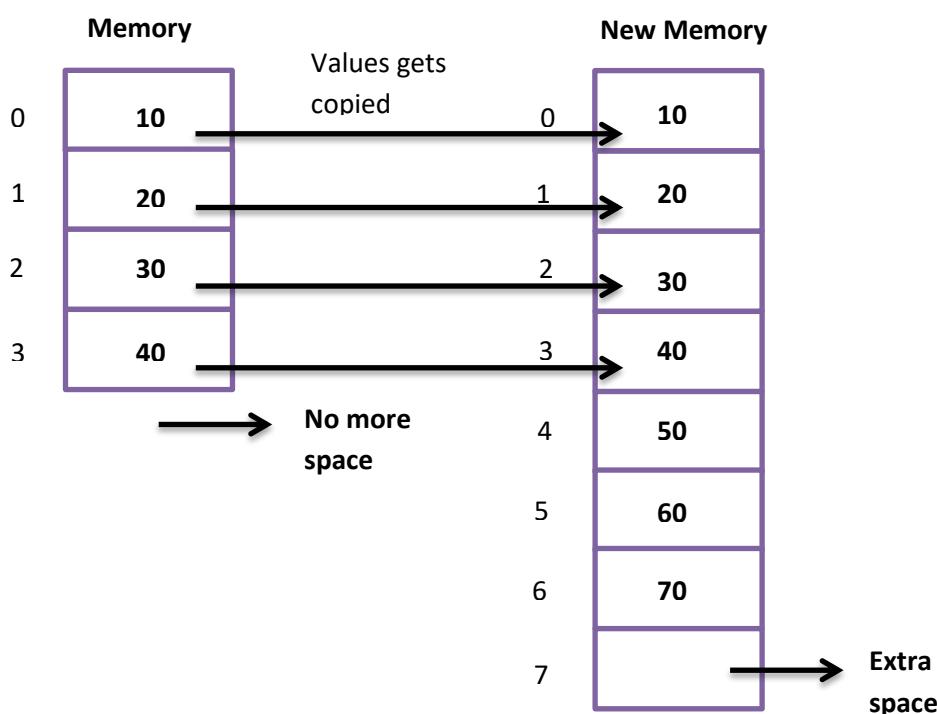
Let us consider we want to store the following data (data here can be heterogeneous as well)

Data: 10 20 30 40 50 60 70

Initially there will be no memory, but let us assume we want to store 10, then instead of giving exact amount of memory it will over allocated the memory, if you want to store one element a list will provide you four memory allocations. This is called as over allocation or amortization which is allocating more than required amount.



Once one by one data is stored in allotted memory at some point there won't be any space left, at that time new array will be created with few more memory allocations. In the new memory the values present earlier are copied and next values will be stored. And the old memory gets de-allocated. Definitely this also comes with its own disadvantages.



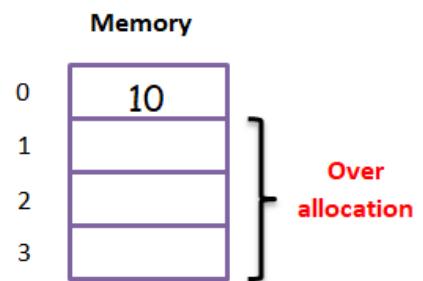
If you are wondering about the resizing pattern then here it is
4, 8, 16, 25, 35, 46, 58, 72...

Earlier we said that when one element is stored 4 memories are allocated, let us see if we can access those memories

```
lst=[]
lst.append(10)
print(lst[0])
```

Output:

```
In [2]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
10
```



10 can be easily accessible, but can the extra memory allocations be?

```
lst=[]
lst.append(10)
print(lst[0])
print(lst[1]) # accessing extra memory
```

Output:

```
File "C:/Users/rooman/OneDrive/Desktop/python/test.py", line 4, in <module>
    print(lst[1]) # accessing extra memory
```

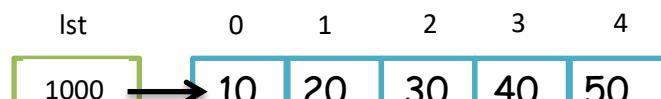


```
IndexError: list index out of range
```

Definitely not! You will get an error stating list index out of range.

Now let us see different ways of creating a list

```
lst=[10,20,30,40,50]
print(lst)
```



Output:

```
In [1]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
[10, 20, 30, 40, 50]
```

Can we only store homogeneous data? Let us see

```
lst=[10,20.5,True,1+3j,"Python"]  
print(lst)
```

Output:



```
In [4]: runfile('C:/Users/rooman/OneDrive/  
Desktop/python/test.py', wdir='C:/Users/rooman/  
OneDrive/Desktop/python')  
[10, 20.5, True, (1+3j), 'Python']
```

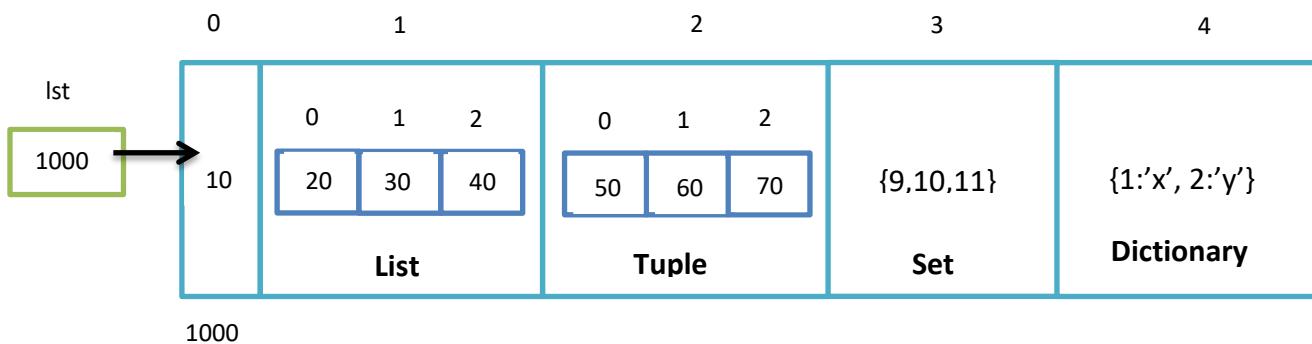
So yes!! We can store heterogeneous data as well. As internally it is stored in referential array format.

Let us see what else can be stored inside a list

```
lst=[10,[20,30,40],(50,60,70),{9,10,11},{1:'x',2:'y'}]  
print(lst)
```

Output:

```
In [5]: runfile('C:/Users/rooman/OneDrive/  
Desktop/python/test.py', wdir='C:/Users/rooman/  
OneDrive/Desktop/python')  
[10, [20, 30, 40], (50, 60, 70), {9, 10, 11},  
{1: 'x', 2: 'y'}]
```



Certainly we can store any type of data and data structures in list.
Now let us try to access individual values in the below example

```

lst=[10,[20,30,40],(50,60,70),{9,10,11},{1:'x',2:'y'}]
print(lst[0])
print(lst[1]) #will print the list in 1st index
print(lst[1][1]) #will print 1st element of list at 1st index
print(lst[2]) #will print tuple
#print(lst[3][1]) #will throw an error as set doesn't have indexes
print(lst[4][2]) #will print the value of 2nd key in dictionary

```

Output:

```

In [7]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
10
[20, 30, 40]
30
(50, 60, 70)
y

```



Operations on list

List can perform two operations

- ❖ Concatenation (using +)
- ❖ Replication (using *)

We have seen these operations in strings, let us see how it works in case of lists.

Concatenation

```

lst1=[10,20,30]
lst2=[40,50,60]
lst3=lst1+lst2 #concatenation
print(lst1)
print(lst2)
print(lst3)

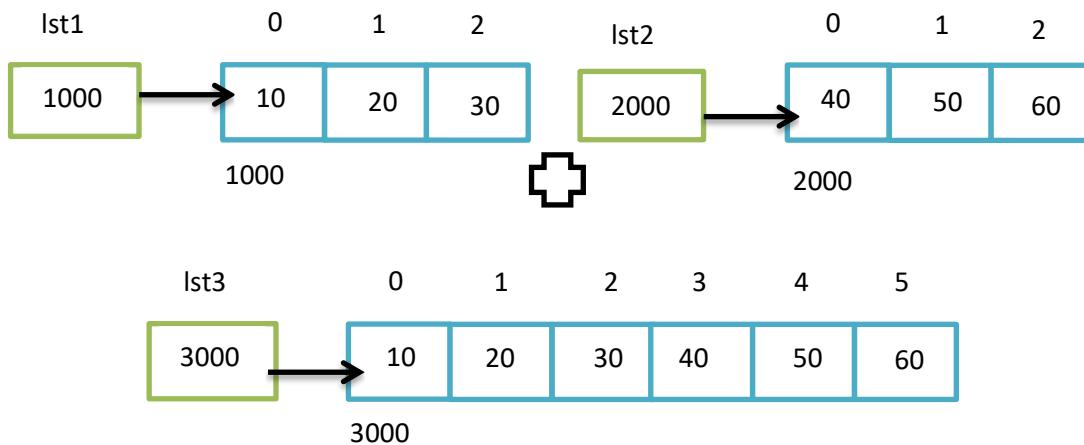
```

Output:

```

In [8]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
[10, 20, 30]
[40, 50, 60]
[10, 20, 30, 40, 50, 60]

```



The order of concatenation matters. If `lst2+lst1` is given the you'll get the output as `[40, 50, 60, 10, 20, 30]`. Why don't you check it yourself?

Replication

Now let us see how to fill entire list with one value

```
lst = [0]*10
print(lst)
```

Output:

```
In [9]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Let us see one more way of using replication

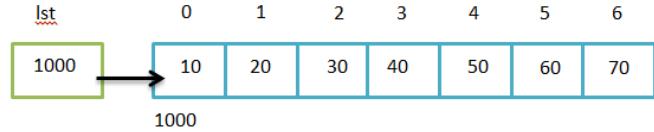
```
lst = [0]*10
print(lst)
lst1 = [1,2,3]*5
print(lst1)
```

Output:

```
In [10]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
```

As we know list is a sequence and so were strings, so let us see if the operations performed on string can be performed with lists as well

```
lst=[10,20,30,40,50,60,70]
print(lst)
print(lst[0])
print(lst[:]) #slicing
print(lst[1:5])
print(lst[::2]) #slicing with step 2
print(lst[-2:-5:-1]) #slicing in reverse
print(lst[::-1]) #printing reverse of list
```



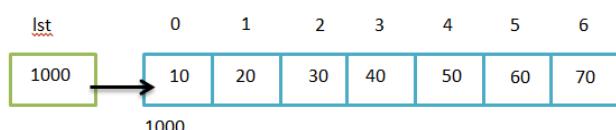
Output:

```
In [11]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
[10, 20, 30, 40, 50, 60, 70]
10
[10, 20, 30, 40, 50, 60, 70]
[20, 30, 40, 50]
[10, 30, 50, 70]
[60, 50, 40]
[70, 60, 50, 40, 30, 20, 10]
```

Slicing in python applies to every sequence type like list, tuple and strings.

Now let us see how loops and lists work together

```
lst=[10,20,30,40,50,60,70]
for i in lst:
    print(i)
```

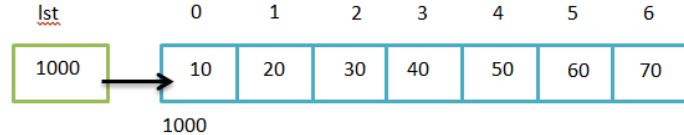


Output:

```
In [12]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
10
20
30
40
50
60
70
```

Now let try to access the values using the index values

```
lst=[10,20,30,40,50,60,70]
for i in range(0,len(lst)):
    print(lst[i])
```



Output:

```
In [13]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
10
20
30
40
50
60
70
```



Python fundamentals

day 26

Today's Agenda

- Mutability of list
- Addition
- Modification
- Removing elements



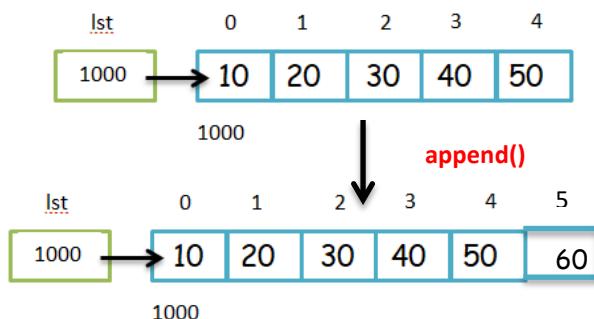
Mutability of list

Let us explore the mutability of list. Mutability of list means add elements to the list, modify elements from the list and remove elements from the list. If these three required are checked then we say that a particular data structure is mutable.

Addition

Let us consider an example and try to add elements

```
lst=[10,20,30,40,50]
print(lst)
lst.append(60) #adding 60 to list
print(lst)
```



Output:

```
In [14]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
[10, 20, 30, 40, 50]
[10, 20, 30, 40, 50, 60]
```

Now let us try to append multiple values

```
lst=[10,20,30,40,50]
print(lst)
lst.append(60) #adding 60 to list
print(lst)
lst.append(70,80,90) #will throw an error
```

Output:

```
File "C:/Users/rooman/OneDrive/Desktop/python/test.py", line 5, in <module>
    lst.append(70,80,90) #will throw an error

TypeError: append() takes exactly one argument
(3 given)
```

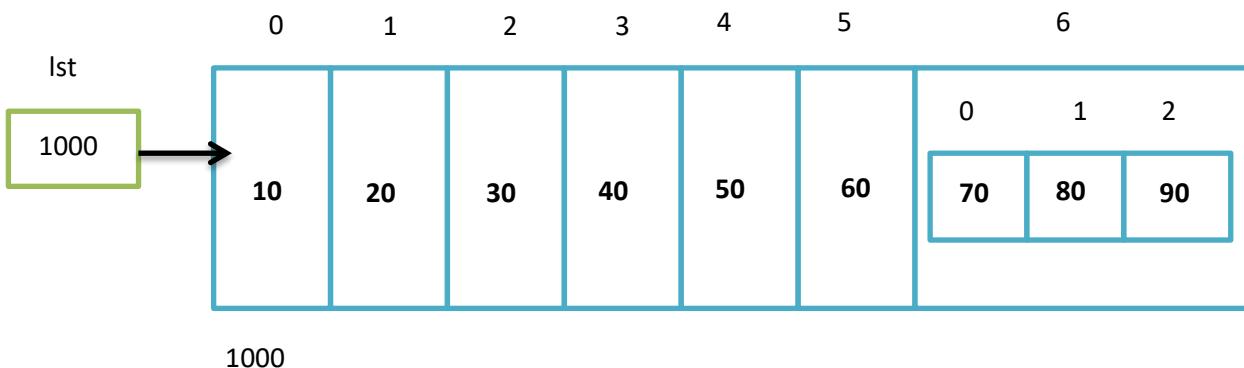


As we can see the error is stating that `append()` can take only one argument. To overcome this we can convert 3 inputs to one like below

```
lst=[10,20,30,40,50]
print(lst)
lst.append(60) #adding 60 to list
print(lst)
lst.append([70,80,90]) #passing 3 inputs as 1 argument
print(lst)
```

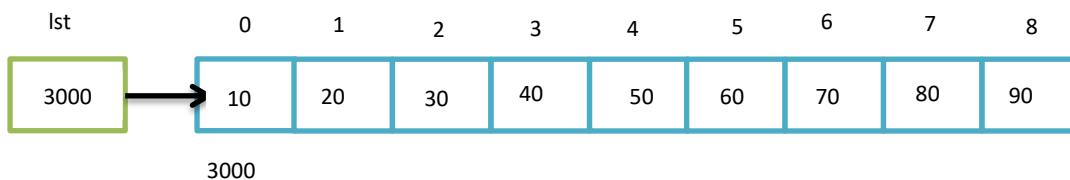
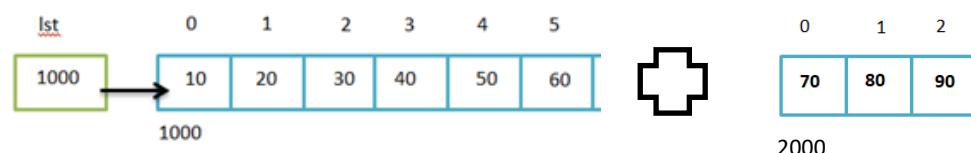
Output:

```
In [17]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
[10, 20, 30, 40, 50]
[10, 20, 30, 40, 50, 60]
[10, 20, 30, 40, 50, 60, [70, 80, 90]]
```



What if we don't want it like this and want to add it individually, let us see

```
lst=[10,20,30,40,50]
print(lst)
lst.append(60) #adding 60 to list
print(lst)
lst = lst + [70,80,90] #concatenation of list to a list
print(lst)
```



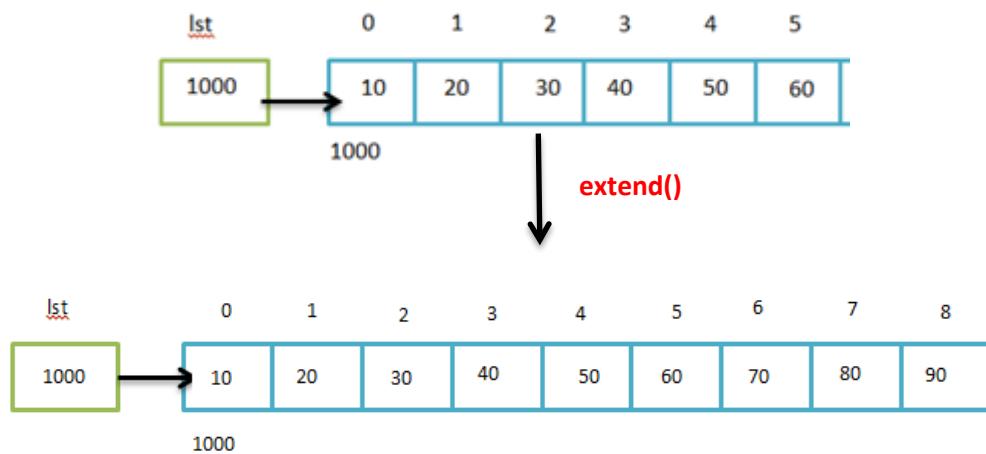
Once the reference is reallocated to new concatenated list, the old ones are dislocated.

Output:

```
In [18]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
[10, 20, 30, 40, 50]
[10, 20, 30, 40, 50, 60]
[10, 20, 30, 40, 50, 60, 70, 80, 90]
```

Even though the above method is providing the expected output, it is not the efficient way. There is a much easier way to achieve the same, let us look at it

```
lst=[10,20,30,40,50]
print(lst)
lst.append(60) #adding 60 to List
print(lst)
lst.extend([70,80,90]) #extends the List
print(lst)
```



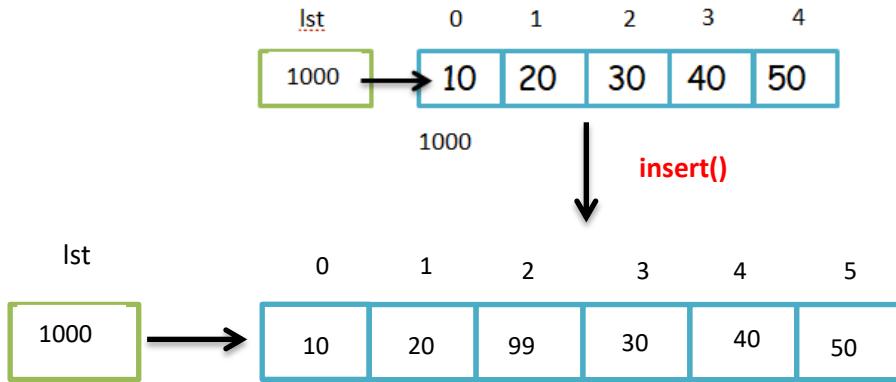
Output:

```
In [20]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
[10, 20, 30, 40, 50]
[10, 20, 30, 40, 50, 60]
[10, 20, 30, 40, 50, 60, 70, 80, 90]
```

Now let us try to insert the elements in middle

```
lst=[10,20,30,40,50]
print(lst)
lst.insert(2,99)
print(lst)
```

`insert()` will take two arguments `index` and `object`. In the specified index if there is no object then the new object gets inserted, if not the size of list increases and all the elements from the specified index are moved towards right and then the new value is inserted.



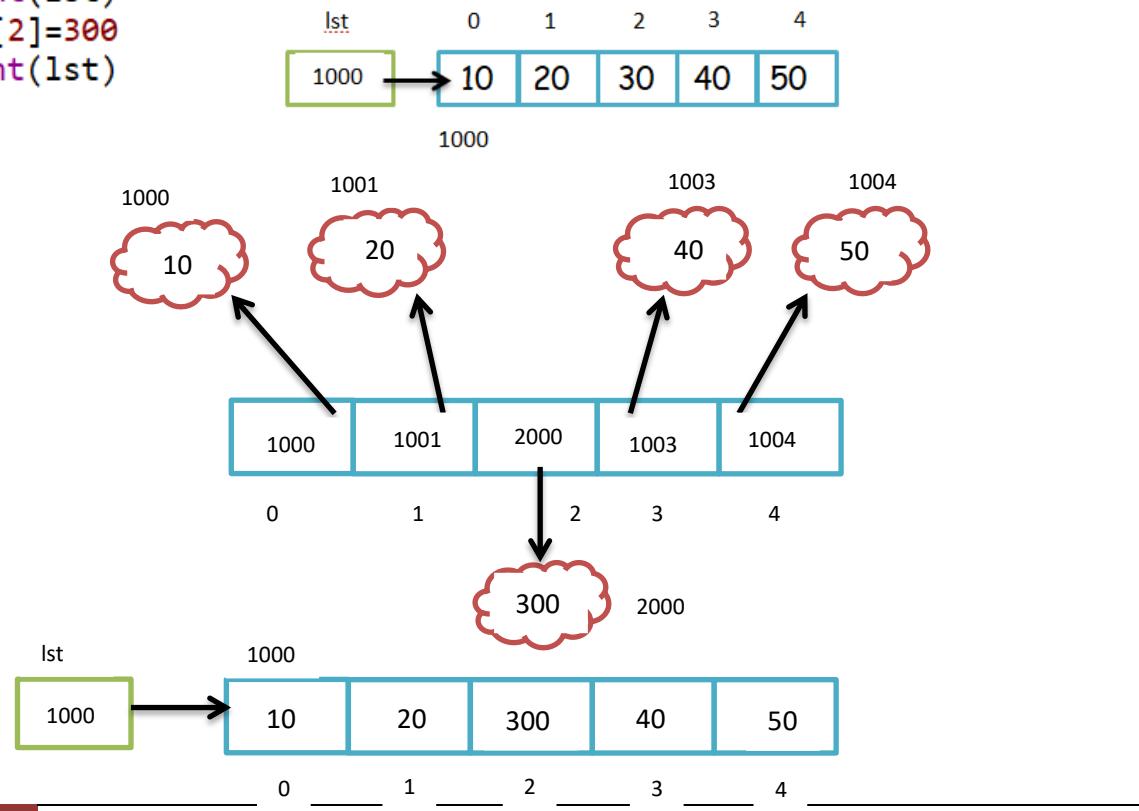
Output:

```
In [21]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
[10, 20, 30, 40, 50]
[10, 20, 99, 30, 40, 50]
```

Modification

Now that we have seen addition, let us see the new feature i.e. modification considering the following example

```
lst=[10,20,30,40,50]
print(lst)
lst[2]=300
print(lst)
```



Output:

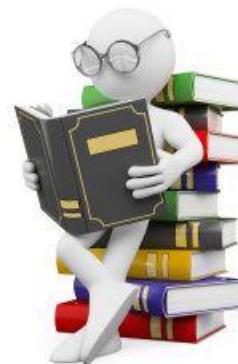
```
In [22]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
[10, 20, 30, 40, 50]
[10, 20, 300, 40, 50]
```

Let us now consider an example where we want to modify a portion of the list by one value

```
lst=[10,20,30,40,50]
print(lst)
lst[1:4]=99
print(lst)
```

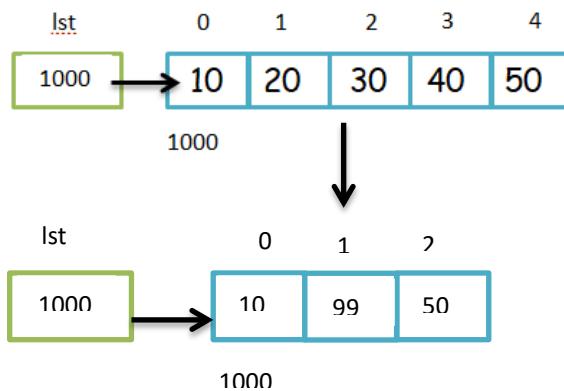
Output:

```
File "C:/Users/rooman/OneDrive/Desktop/python/test.py", line 3, in <module>
    lst[1:4]=99
TypeError: can only assign an iterable
```



Whenever we are doing slicing assignment, we should always assign an iterable. 99 can be easily converted to an iterable by placing it within [] like below

```
lst=[10,20,30,40,50]
print(lst)
lst[1:4]=[99]
print(lst)
```



Output:

```
In [24]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
[10, 20, 30, 40, 50]
[10, 99, 50]
```

Now let us try to modify multiple values

```
lst=[10,20,30,40,50]
print(lst)
lst[1:4]=[99,88,77]
print(lst)
```



Output:

```
In [25]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
[10, 20, 30, 40, 50]
[10, 99, 88, 77, 50]
```

But now what if we want to add more than three values to a slice of three values? Let us see

```
lst=[10,20,30,40,50]
print(lst)
lst[1:4]=[99,88,77,66]
print(lst)
```

Output:

```
In [26]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
[10, 20, 30, 40, 50]
[10, 99, 88, 77, 66, 50]
```

Certainly even that is possible. We can add as many elements as we want in certain portion of list.

Now our expectation is to modify every alternative element by a value

```
lst=[10,20,30,40,50]
print(lst)
lst[::2]=[99,99,99]
print(lst)
```

Output:

```
In [27]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
[10, 20, 30, 40, 50]
[99, 20, 99, 40, 99]
```

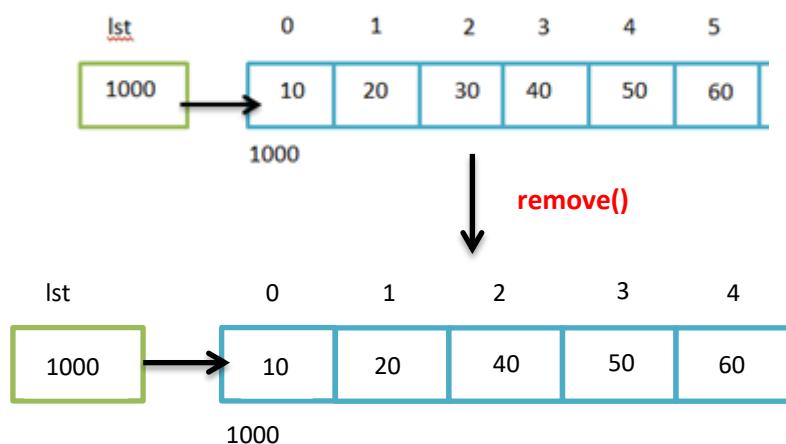


Removing elements

Now that we have seen addition of elements, modification of elements, let us see the last but not the least removing elements.

Consider the following example

```
lst=[10,20,30,40,50,60]
print(lst)
lst.remove(30)
print(lst)
```



Output:

```
In [28]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
[10, 20, 30, 40, 50, 60]
[10, 20, 40, 50, 60]
```

But what if we have two same values which will get removed? Let us see in below example

```
lst=[10,20,30,40,50,30]
print(lst)
lst.remove(30)
print(lst)
```

Output:



```
In [29]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
[10, 20, 30, 40, 50, 30]
[10, 20, 40, 50, 30]
```

Always the first occurrence is going to be removed. But what if we want to remove all occurrences of the value? But before we do that we should first check for that value. So let us see below how to do it

```
lst=[10,20,30,40,50,30]
print(30 in lst) #checking for presence of object
print(30 not in lst) #checking for absence of object
```

Output:

```
In [30]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
True
False
```

So now that we know how to check presence/absence of an object we shall next see how to remove all the occurrences

```
lst=[10,20,30,40,50,30]
print(lst)
while 30 in lst:
    lst.remove(30)
print(lst)
```

We know that whenever we are not aware of how many times an operation must be performed, **while** loop should be used.

Output:

```
In [31]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
[10, 20, 30, 40, 50, 30]
[10, 20, 40, 50]
```



Python Fundamentals

day 27

Today's Agenda

- Removing elements (contd)
- Creation a copy of list
- Nested list



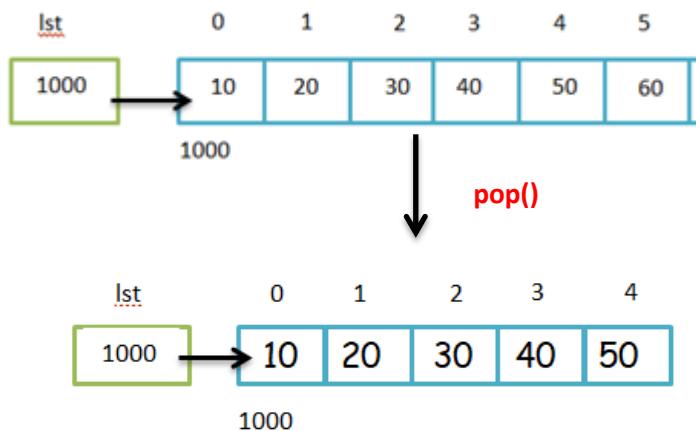
Removing elements (contd)

In previous session we have seen different ways to remove an element. Let us continue to explore some more methods

`pop()`

`pop()` method by default pop the last element from the list as shown below

```
lst=[10,20,30,40,50,60]
print(lst)
lst.pop()
print(lst)
```

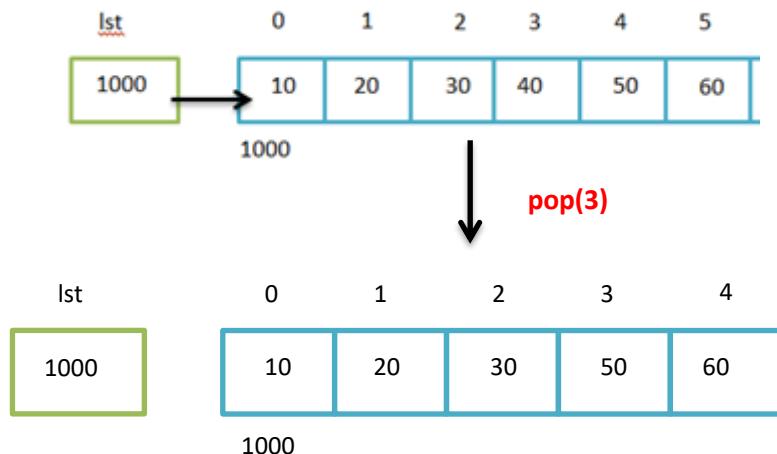


Output:

```
In [32]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
[10, 20, 30, 40, 50, 60]
[10, 20, 30, 40, 50]
```

Now what if we want to pop an element in between the list, let us see how to do it

```
lst=[10,20,30,40,50,60]
print(lst)
lst.pop(3)
print(lst)
```



Output:

```
In [33]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
[10, 20, 30, 40, 50, 60]
[10, 20, 30, 50, 60]
```

del

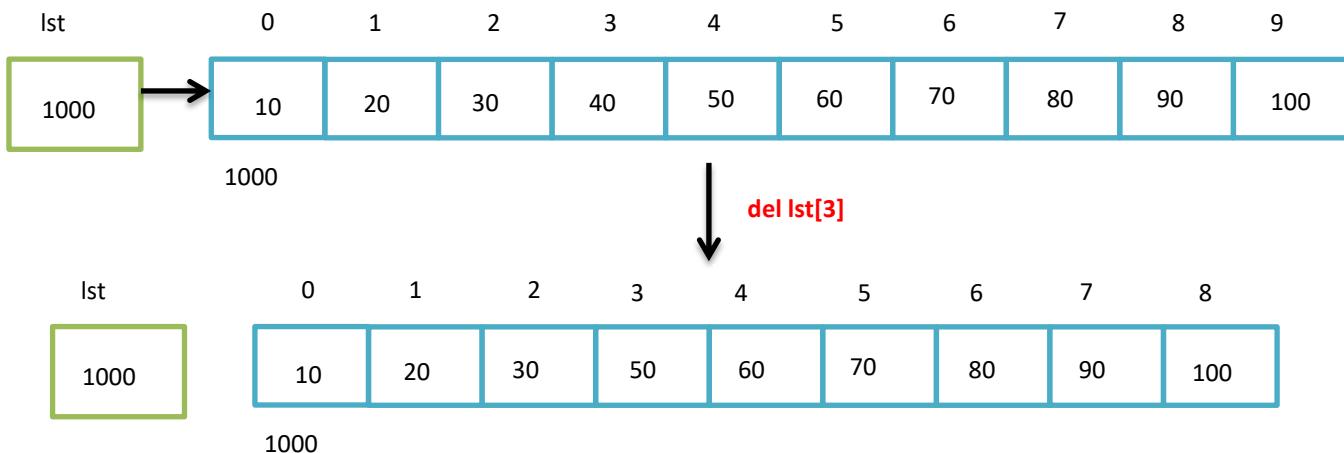
del is another keyword using which you can delete an elements from the list by mentioning its index

Let us consider the following example and get to know it in better way

```

lst=[10,20,30,40,50,60,70,80,90,100]
print(lst)
del lst[3]
print(lst)

```



Output:

```

In [34]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
[10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
[10, 20, 30, 50, 60, 70, 80, 90, 100]

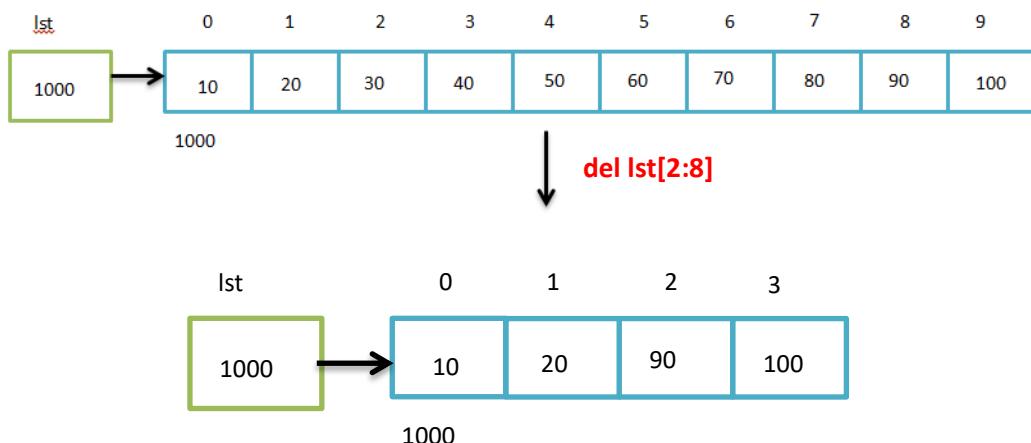
```

Now let us see how to **delete a portion** of the list

```

lst=[10,20,30,40,50,60,70,80,90,100]
print(lst)
del lst[2:8] # deleting a portion from the list
print(lst)

```

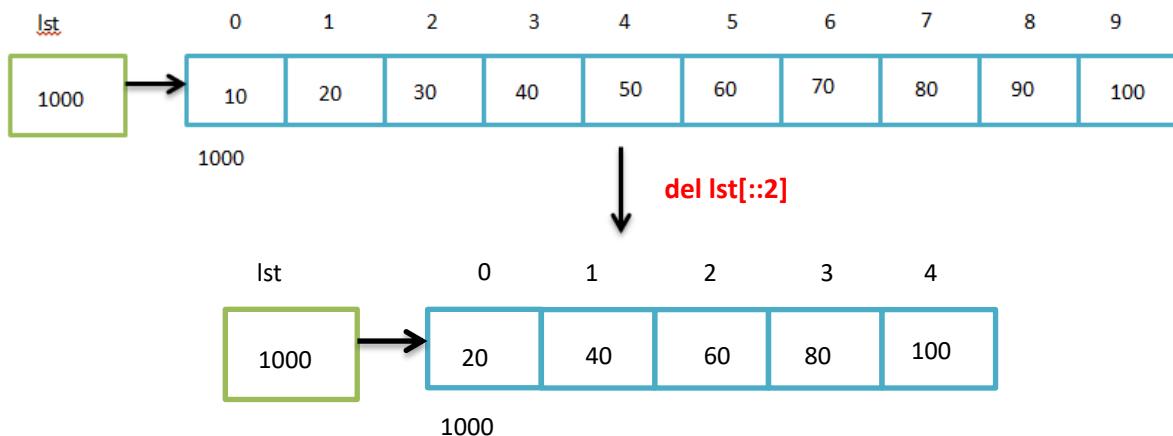


Output:

```
In [35]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
[10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
[10, 20, 90, 100]
```

Now let's try to **delete alternative elements** from the same list

```
lst=[10,20,30,40,50,60,70,80,90,100]
print(lst)
del lst[::2] # deleting alternative elements
print(lst)
```



Output:

```
In [36]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
[10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
[20, 40, 60, 80, 100]
```

Now let us try deleting a portion of elements in reverse order

```
lst=[10,20,30,40,50,60,70,80,90,100]
print(lst)
del lst[-4:-9:-1] # deleting in reverse order
print(lst)
```

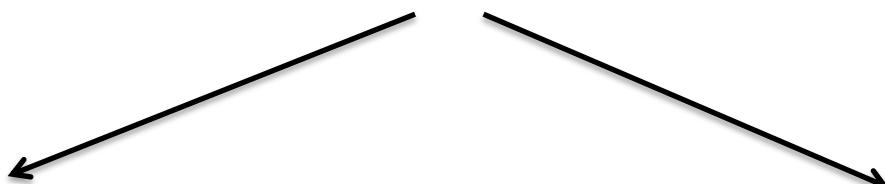
Consider we want to delete elements from -4 to -9 in reverse direction i.e. -1



Output:

```
In [37]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
[10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
[10, 20, 80, 90, 100]
```

Deletion of elements from a list



Based on values

remove() - will delete elements based on values.

Based on index

pop() - will delete elements based on indexes, but cannot delete more than one value.

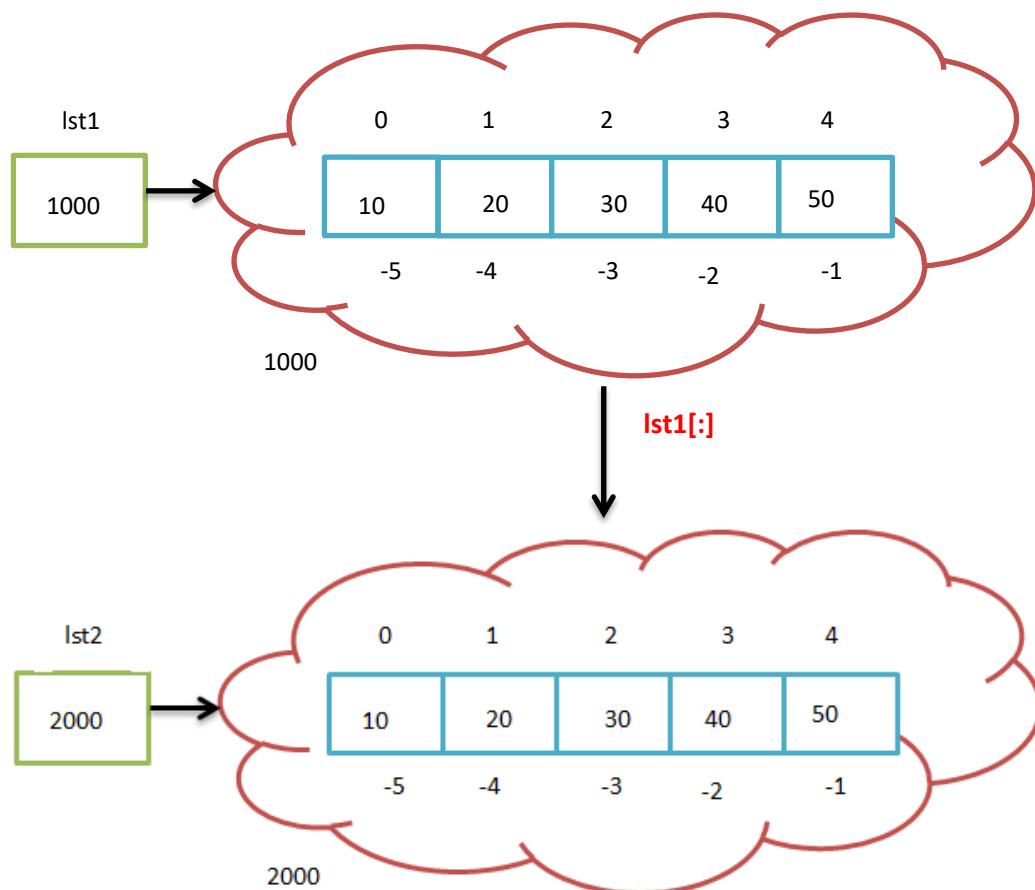
del keyword helps in deleting more than one element using its index values.



Creation a copy of list

Creating a copy of list in python is super easy, let us see how

```
lst1=[10,20,30,40,50]
lst2=lst1[:] #creation a copy of lst1
print(lst1)
print(lst2)
print(lst1 is lst2)
```



Output:

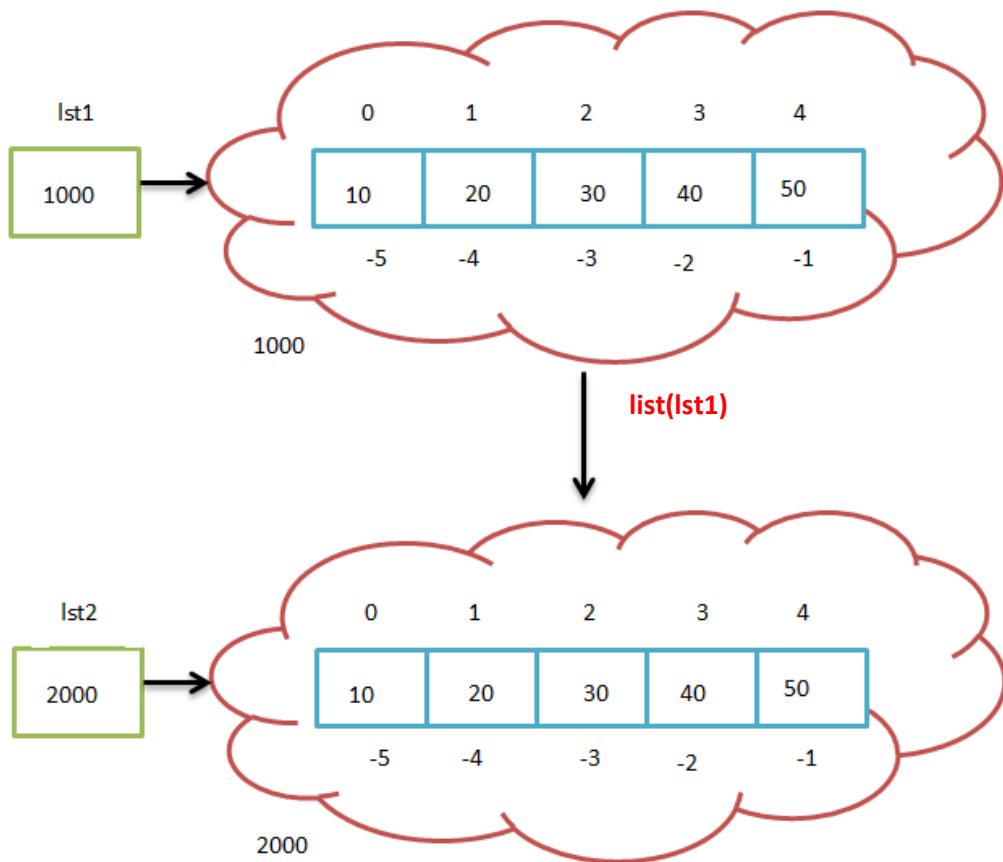
```
In [38]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
[10, 20, 30, 40, 50]
[10, 20, 30, 40, 50]
False
```

There is also another way to create a copy, let us see how

```

lst1=[10,20,30,40,50]
lst2=list(lst1) #creation a copy of lst1 using list()
print(lst1)
print(lst2)
print(lst1 is lst2)

```



Output:

```

In [39]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
[10, 20, 30, 40, 50]
[10, 20, 30, 40, 50]
False

```

Nested list

Just like nested loops, nested list means presence of lists within a list. Let us see how to copy nested lists and by any means do they differ

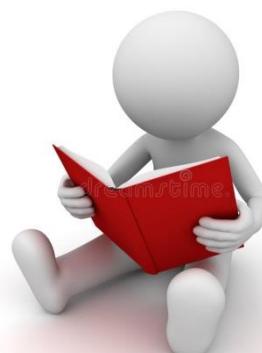
Let us consider the following example where we are trying to create a copy of list `lst1` and if made changes in original list `lst1` the copy `lst2` shouldn't get affected

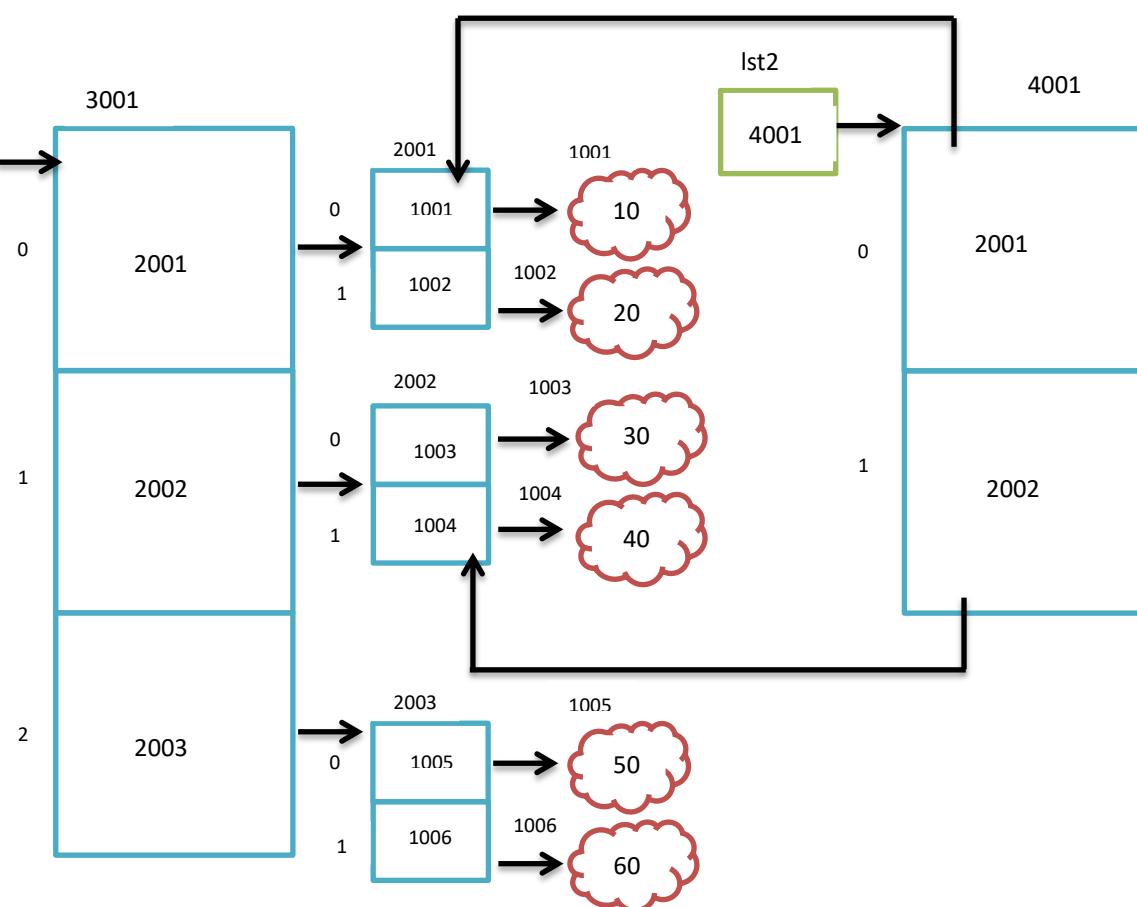
```
lst1=[[10,20],[30,40]]  
lst2=list(lst1)  
print(lst1)  
print(lst2)  
print(lst1 is lst2)  
lst1.append([50,60])#adding elements to original list  
print(lst1)  
print(lst2)  
lst1[1][0]=300 #modifying value of original list  
print(lst1)  
print(lst2) #changes shouldn't have reflected
```

Output:

```
In [40]: runfile('C:/Users/rooman/OneDrive/  
Desktop/python/test.py', wdir='C:/Users/rooman/  
OneDrive/Desktop/python')  
[[10, 20], [30, 40]]  
[[10, 20], [30, 40]]  
False  
[[10, 20], [30, 40], [50, 60]]  
[[10, 20], [30, 40]]  
[[10, 20], [300, 40], [50, 60]]  
[[10, 20], [300, 40]]
```

If tried to modify the values in original list `lst1` the so called copy of list `lst2` also got reflected. Which is not what we expected, so let us see what happens through memory perspective





Above is the memory perspective of the given example. `lst2` is the shallow copy of `lst1`. A shallow copy constructs a new compound object and then (to the extent possible) inserts references into it to the objects found in the original.

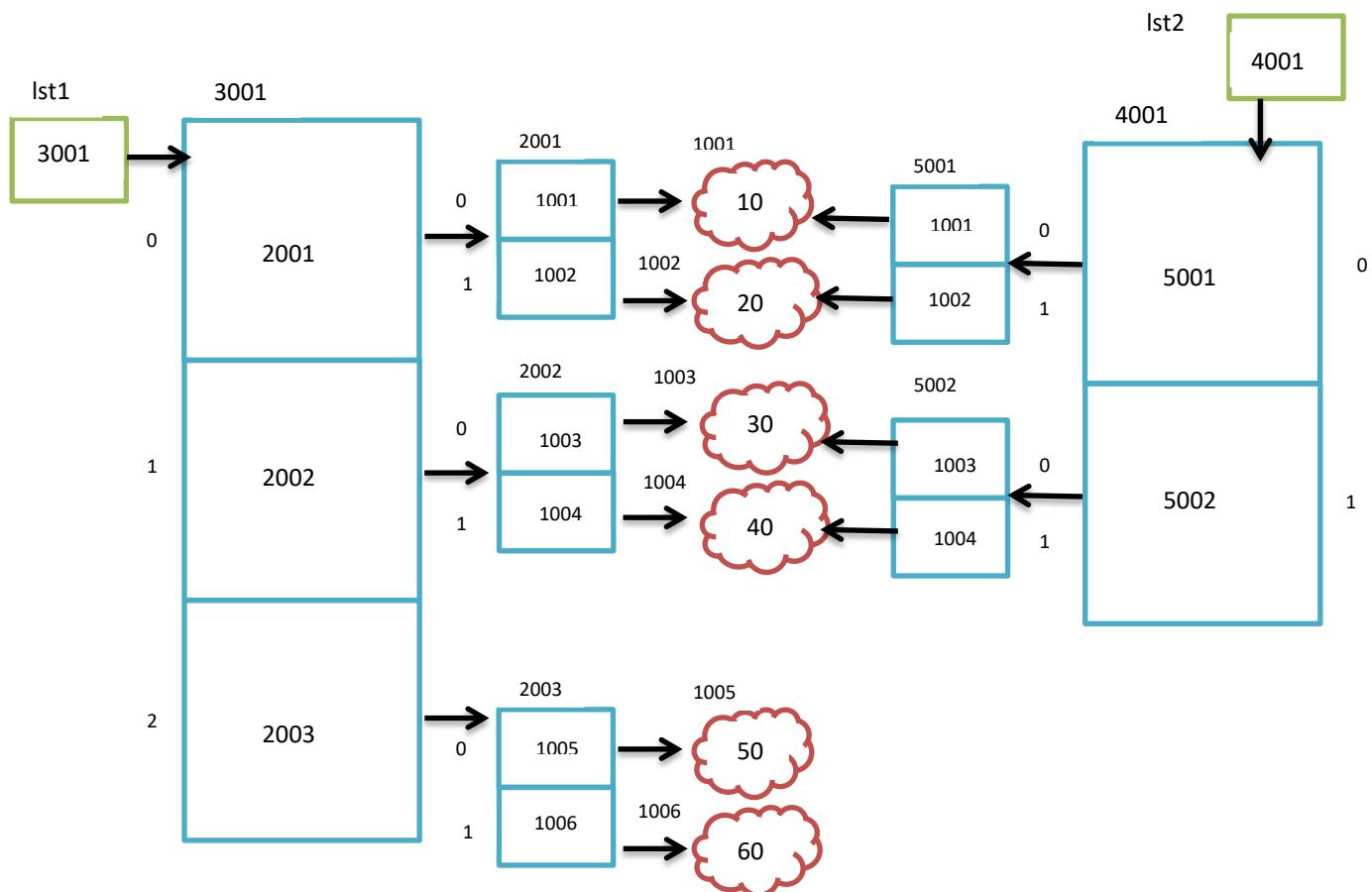
To overcome the result of shallow copy we have something called as **deep copy** which is present in **copy module**. Let us see how to make use of it in below example

```
import copy
lst1=[[10,20],[30,40]]
lst2=copy.deepcopy(lst1) #creating deepcopy
print(lst1)
print(lst2)
print(lst1 is lst2)
lst1.append([50,60])#adding elements to original list
print(lst1)
print(lst2)
lst1[1][0]=300 #modifying value of original list
print(lst1)
print(lst2) #changes shouldn't have reflected
```

Output:

```
In [41]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
[[10, 20], [30, 40]]
[[10, 20], [30, 40]]
False
[[10, 20], [30, 40], [50, 60]]
[[10, 20], [30, 40]]
[[10, 20], [300, 40], [50, 60]]
[[10, 20], [30, 40]]
```

Great!! Now the output is as we expected, but let us see from memory perspective what has changed



A **deep copy** constructs a new compound object and then, recursively, inserts **copies** into it of the objects found in the original.

Python Fundamentals

day 28

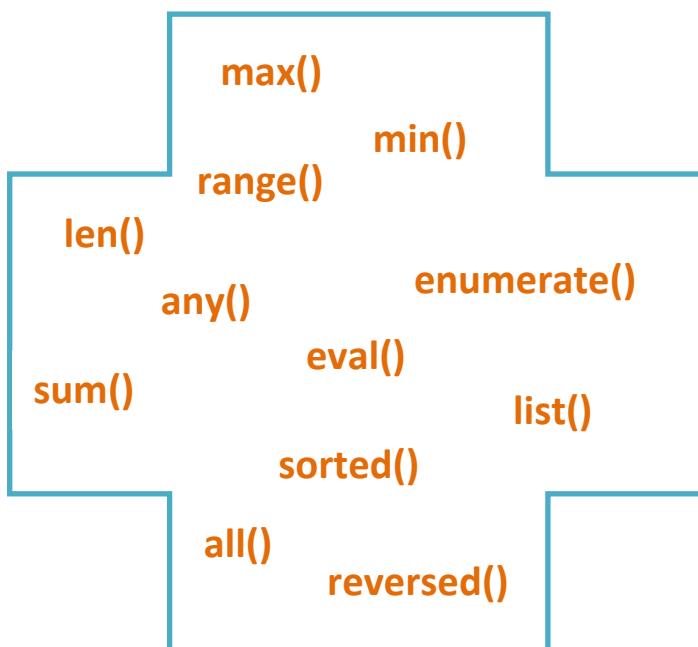
Today's Agenda

- Built-in functions (lists)
- Methods
- Programs



Built-in functions (lists)

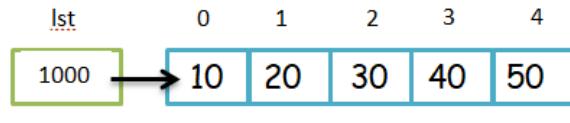
Any **function** that is provided as part of a high-level language and can be executed by a simple reference with specification of arguments.



Let us get to know the need of these built-in functions one by one

Consider the following example

```
lst=[10,20,30,40,50]
print(len(lst)) #prints length of list
print(max(lst)) #prints maximum value of list
print(min(lst)) #prints minimum value of list
print(sum(lst)) #prints sum of all elements in list
```



Output:

```
In [1]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
5
50
10
150
```

The below built-in functions will be using while looping a list or iterate a list

```
lst=[10,20,30,40,50]

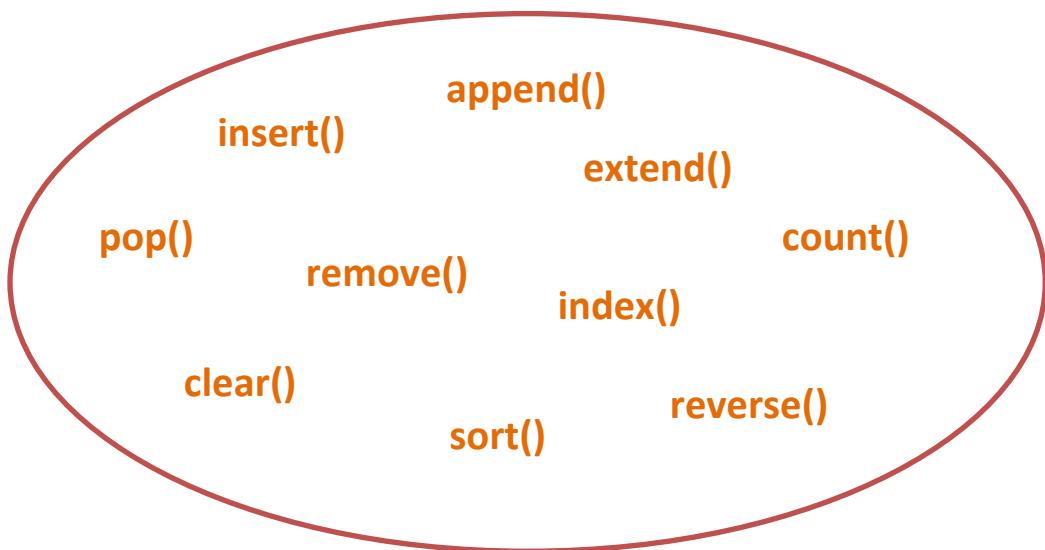
for i,j in enumerate(lst):
    print(i,j) #gives index and the value associated to it
```

Output:

```
In [2]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
0 10
1 20
2 30
3 40
4 50
```

Methods

Everything in python is an object, which means everything has its own type and type is technically referred to as class. And the functions present within a class are called as methods.



Let us see some examples using these methods

```
lst=[10,20,30,40,50,20]

print(lst.count(20)) #gives the count of value inserted
print(lst.index(40)) #gives the index of value inserted
print(lst.index(20)) #gives the first occurrence index
print(lst.index(20,2,6)) #specifying value and its search of index in a range
```

Output:

```
In [4]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
2
3
1
5
```

Next example is to show how to delete all the elements from list

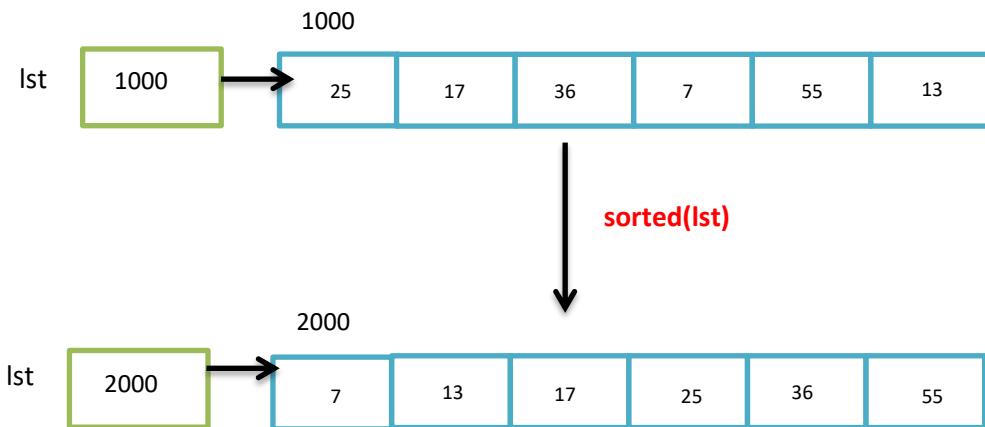
```
lst=[10,20,30,40,50,20]
print(lst)
lst.clear()
print(lst)
```

Output:

```
In [5]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
[10, 20, 30, 40, 50, 20]
[]
```

Now our expectation is to arrange the elements in certain order. Let us see how to achieve it

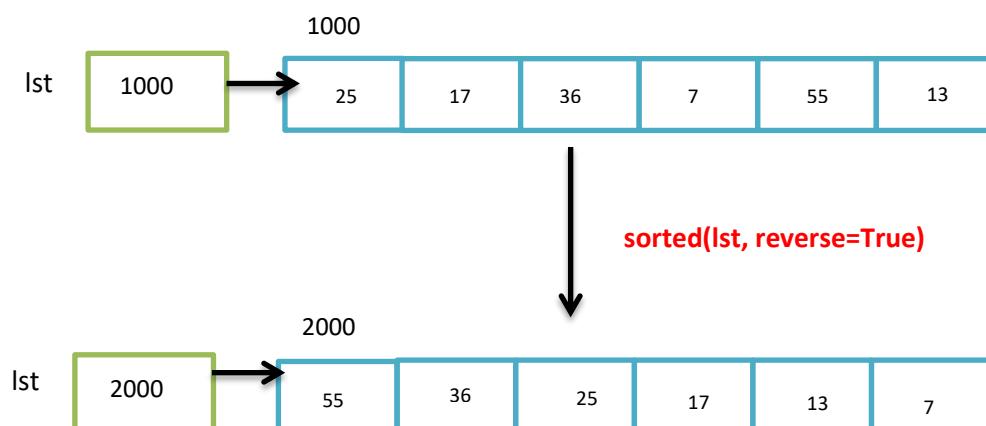
```
lst=[25,17,36,7,55,13]
print(lst)
lst=sorted(lst) #by default arranges in ascending order
print(lst)
```



Output:

```
In [7]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
[25, 17, 36, 7, 55, 13]
[7, 13, 17, 25, 36, 55]
```

Now if we want the same in descending order



```
lst=[25,17,36,7,55,13]
print(lst)
lst=sorted(lst, reverse=True) #arranges in descending order
print(lst)
```

Output:

```
In [8]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
[25, 17, 36, 7, 55, 13]
[55, 36, 25, 17, 13, 7]
```

sorted() was the built-in function. But we also have a method which can perform the same operation. Let us see

```
lst=[25,17,36,7,55,13]
print(lst)
lst.sort() #arranges in ascending order
print(lst)
```

Output:

```
In [10]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
[25, 17, 36, 7, 55, 13]
[7, 13, 17, 25, 36, 55]
```

If descending order is what we want, then

```
lst=[25,17,36,7,55,13]
print(lst)
lst.sort(reverse=True) #arranges in descending order
print(lst)
```

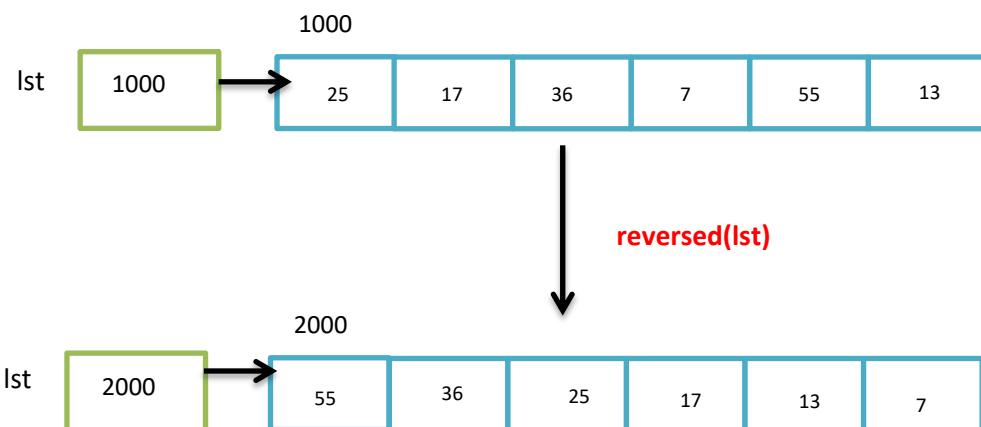
Output:

```
In [9]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
[25, 17, 36, 7, 55, 13]
[55, 36, 25, 17, 13, 7]
```



The main difference between `sorted()` and method `sort()` is that in `sorted()` a new copy of list is created where the sorting is done and new reference needs to be given. That means extra memory. But in `sort()` the sorting happens within the same list.

Now we want to reverse the list



```
lst=[25,17,36,7,55,13]
print(lst)
lst_rev = list(reversed(lst)) #reversed will give an object of reversediterator
print(lst_rev)
```

Output:

```
In [11]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
[25, 17, 36, 7, 55, 13]
[13, 55, 7, 36, 17, 25]
```

But is there a method which can do the same operation in an efficient way? Definitely there is

```
lst=[25,17,36,7,55,13]
print(lst)
lst.reverse()
print(lst)
```

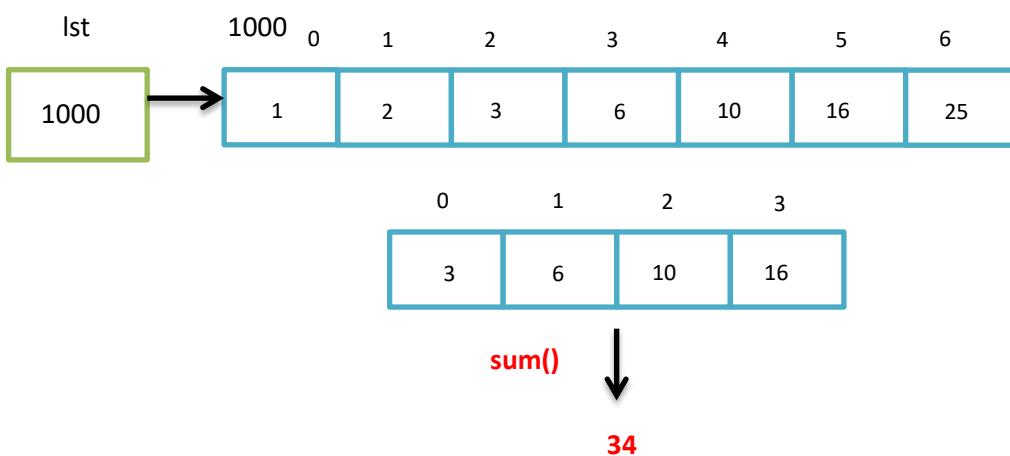
Output:

```
In [12]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
[25, 17, 36, 7, 55, 13]
[13, 55, 7, 36, 17, 25]
```

Programs

Example 1: Program to find sum of sublist.

```
lst=input("Enter a list between []\n")
lst=eval(lst) #converts string into list
start=int(input("Enter the start index"))
stop=int(input("Enter the stop index"))
print("sum =",sum(lst[start:stop+1]))
```



Example 2: Program to append the elements which is not present in the primary list

```
lst1=[10,20,30,40]
lst2=[35,20,10,15]

for i in lst2:
    if i not in lst1:
        lst1.append(i)

print(lst1)
```

Output:

```
In [15]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
[10, 20, 30, 40, 35, 15]
```



Python Fundamentals

day 29

Today's Agenda

- Programs on list



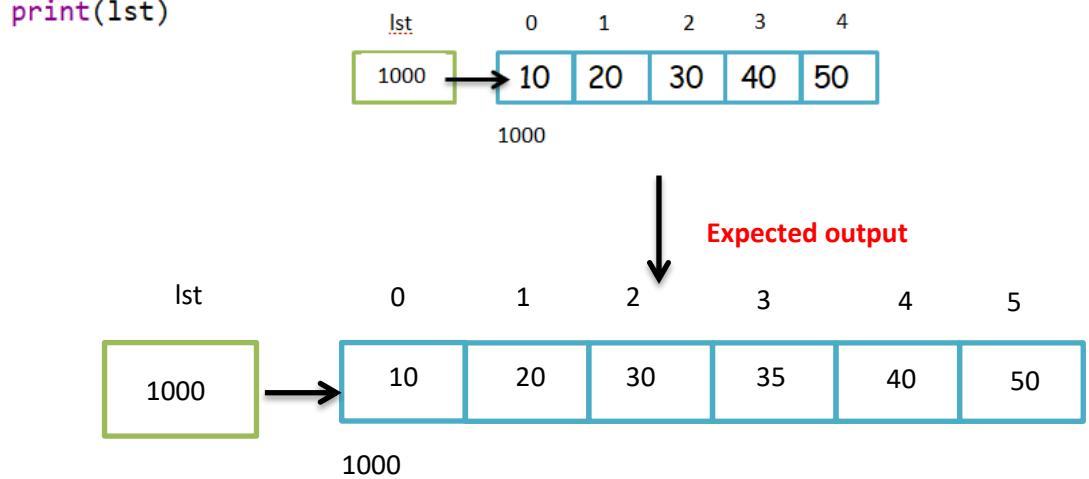
Programs

Example 1: Program to insert an element at right position within a sorted list.

```
lst=eval(input("Enter a sorted list between []\n"))
n=int(input("Enter the value to be inserted\n"))
print(lst)

for i in range(len(lst)):
    if n<lst[i]:
        lst.insert(i,n)
        break

print(lst)
```



Output:

```
In [1]: runfile('C:/Users/rooman/OneDrive/  
Desktop/python/test.py', wdir='C:/Users/rooman/  
OneDrive/Desktop/python')
```

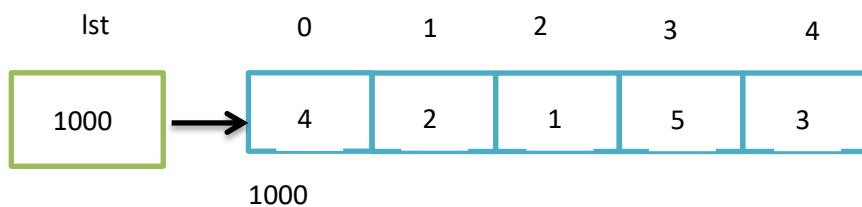
```
Enter a sorted list between []  
[10,20,30,40,50]
```

```
Enter the value to be inserted  
35  
[10, 20, 30, 40, 50]  
[10, 20, 30, 35, 40, 50]
```

Example 2: Given five positive integers, find the minimum and maximum values that can be calculated by summing exactly four of the five integers. Then print the respective minimum and maximum values as a single line of two space-separated long integers.

For example, `arr[1,3,5,7,9]`. Our minimum sum is $1+3+5+7=16$ and our maximum sum is $3+5+7+9=24$. We would print 16 24

Let us consider the following case



Missing
number

$$15 = 3 + 4 + 2 + 1 + 5 = 12$$

$$\text{sum(lst)} - \max(\text{lst}) \quad 15 = 5 + 4 + 2 + 1 + 3 = 10 \quad \text{min}$$

$$\text{sum(lst)} - \min(\text{lst}) \quad 15 = 1 + 4 + 2 + 5 + 3 = 14 \quad \text{max}$$

$$15 = 2 + 4 + 1 + 5 + 3 = 13$$

$$15 = 4 + 2 + 1 + 5 + 3 = 11$$



```
lst=[4,2,1,5,3]
print(sum(lst)-max(lst),sum(lst)-min(lst))
```

Output:

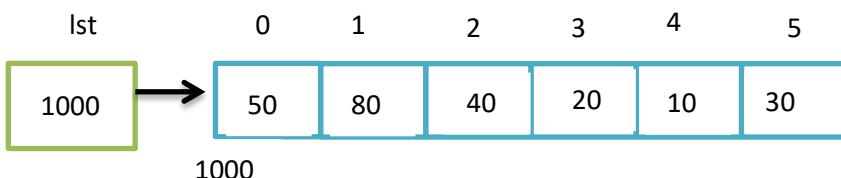
```
In [2]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
10 14
```

Example 3: Find the sum of minimum number and maximum number of a list without using any built-in functions

```
lst=[50,80,40,20,10,30]
min=lst[0]
max=lst[0]

for i in range(len(lst)):
    if lst[i]<min:
        min=lst[i]
    elif lst[i]>max:
        max=lst[i]

print("Min + Max =",min+max)
```



$$\text{minimum} = 10 \quad \text{maximum} = 80$$

$$\text{Sum} = \text{minimum} + \text{maximum}$$

$$\text{Sum} = 10 + 80 = 90$$

Output:

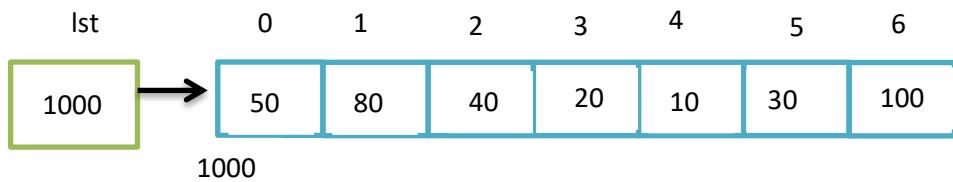
```
In [3]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
Min + Max = 90
```

Example 4: To calculate the length of list without using built-in functions

```
lst=[50,80,40,20,10,30,100]
count=0

for i in lst:
    count+=1

print(f"Length = {count}")
```



Output:

```
In [5]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
Length = 7
```

Example 5: To calculate the sum of values of entire list without using built-in method.

```
lst=[50,80,40,20,10,30,100]
sum=0

for i in lst:
    sum+=i

print(f"sum = {sum}")
```



Output:

```
In [6]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
sum = 330
```

Example 6: Similarly calculate product of all elements.

```
lst=[50,80,40,20,10,30,100]
p=1

for i in lst:
    p=p*i

print(f"Product = {p}")
```

Output:

```
In [10]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
Product = 9600000000
```

Try yourself what happens if the list is empty?

And find a solution for it.



Python Fundamentals

day 30

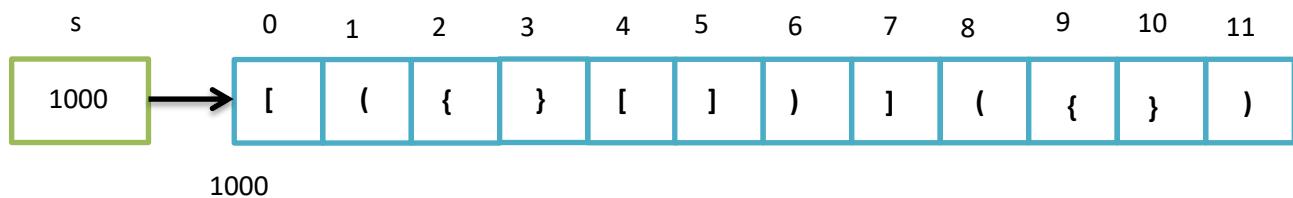
Today's Agenda

- Programs on lists
- List comprehension



Program on list

Example 1: Program to check if the expression contains balanced parenthesis.



```
s=input("Enter the an expression within brakets\n")
lst=[]
for i in s:
    if i=='[' or i=='(' or i=='{':
        lst.append(i)
    elif i==']' and lst[-1]=='[':
        lst.pop()
    elif i==')' and lst[-1]=='(':
        lst.pop()
    elif i=='}' and lst[-1]=='{':
        lst.pop()
    else:
        break

if len(lst)==0:
    print(s,"Expression is balanced")
else:
    print(s,"Expression is not balanced")
```

Output:

```
In [5]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
```

Enter the an expression within brakets
[({}[])]({})
[({}[])]({}) Expression is balanced

```
In [6]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
```

Enter the an expression within brakets
[(({})]
[(({})] Expression is not balanced

Example 2: Comparison of lists.



```
lst1=[7,3,5]
lst2=[7,3,5]
print(lst1==lst2)
```

Output:

```
In [7]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
True
```

```
lst1=[7,3,5]
lst2=[7,3,5]
print(lst1!=lst2)
```



Output:

```
In [9]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
False
```



```
lst1=[7,3,5]
lst2=[7,13,5]
print(lst1==lst2)
```

Output:

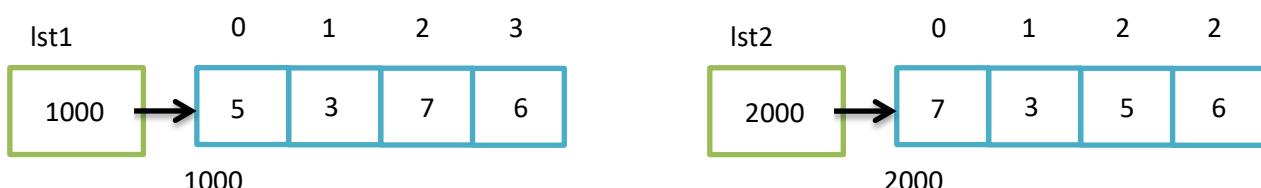
```
In [8]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
False
```

```
lst1=[7,3,5]
lst2=[7,13,5]
print(lst1!=lst2)
```



Output:

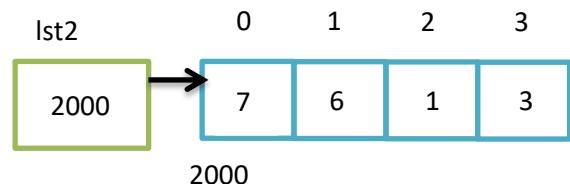
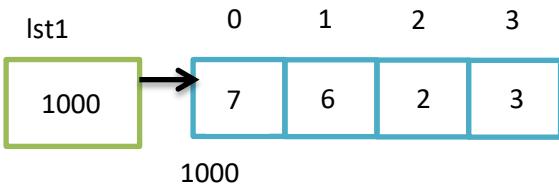
```
In [10]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
True
```



```
lst1=[5,3,7,6]
lst2=[7,3,5,6]
print(lst1==lst2)
```

Output:

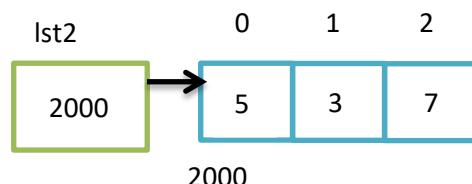
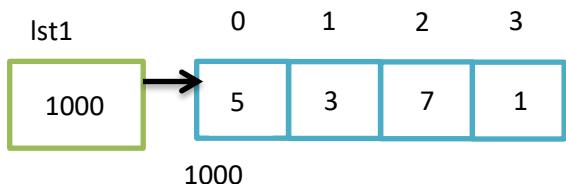
```
In [11]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
False
```



```
lst1=[7,6,2,3]
lst2=[7,6,1,3]
print(lst1>lst2)
```

Output:

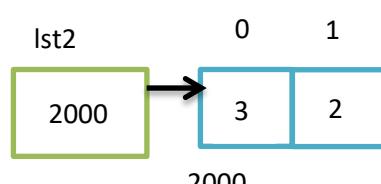
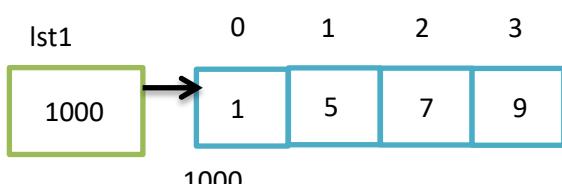
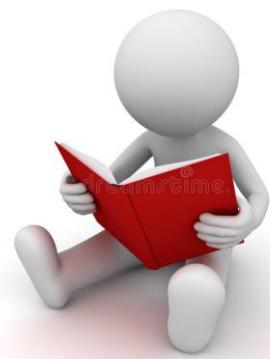
```
In [12]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
True
```



```
lst1=[5,3,7,1]
lst2=[5,3,7]
print(lst1>lst2)
```

Output:

```
In [13]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
True
```



```
lst1=[1,5,7,9]
lst2=[3,2]
print(lst1>lst2)
```

Output:

```
In [14]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
False
```

List Comprehension

Before knowing what is list comprehension let us consider an example and later on see how the same example can be written in much simpler way using list comprehension

```
lst=[2,5,7,8,10,12]
sq_lst=[]

for i in lst:
    sq_lst.append(i**2)

print(lst)
print(sq_lst)
```

Output:

```
In [1]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
[2, 5, 7, 8, 10, 12]
[4, 25, 49, 64, 100, 144]
```



Now let us see the same example with list comprehension

Syntax:

[What I want From which loop **On what condition**] //non-technical

[**Expression for item in list If condition**] //technical



Optional

```
lst=[2,5,7,8,10,12]
sq_lst=[i**2 for i in lst]

print(lst)
print(sq_lst)
```

Output:

```
In [2]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
[2, 5, 7, 8, 10, 12]
[4, 25, 49, 64, 100, 144]
```

Now let us consider an example where the condition is used. The condition is, instead of printing squares of all the numbers let us print only even numbers square in both traditional way and using list comprehension

```
lst=[2,5,7,8,10,12]
sq_lst=[]

for i in lst:
    if i%2==0:
        sq_lst.append(i**2)

print(lst)
print(sq_lst)
```

Output:

```
In [3]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
[2, 5, 7, 8, 10, 12]
[4, 64, 100, 144]
```

Now let us see how the same code changes when used list comprehension



```
lst=[2,5,7,8,10,12]
sq_lst=[i**2 for i in lst if i%2==0]

print(lst)
print(sq_lst)
```

Output:

```
In [4]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
[2, 5, 7, 8, 10, 12]
[4, 64, 100, 144]
```



Python fundamentals

day 32

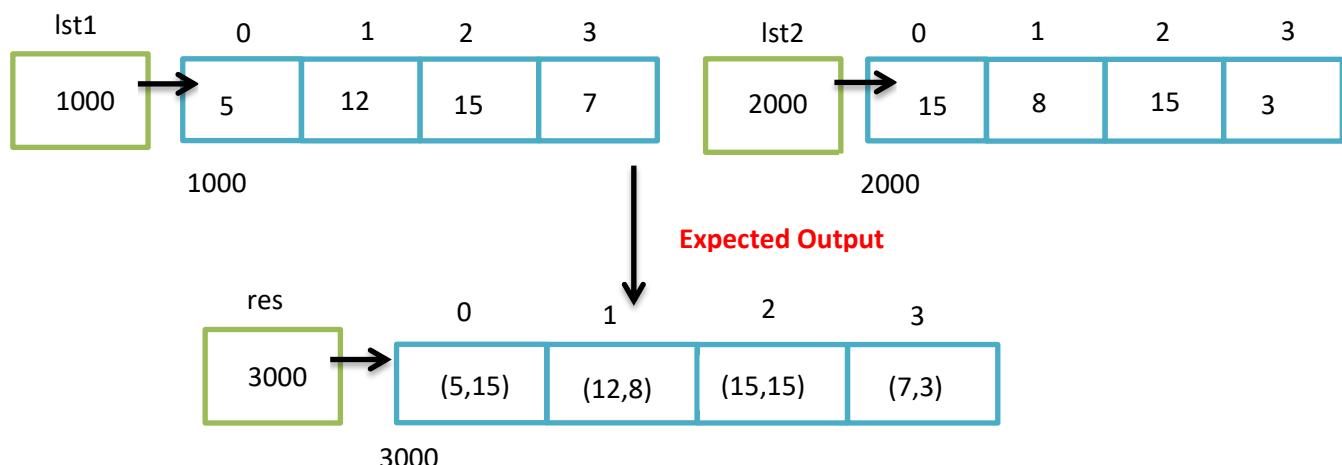
Today's Agenda

- Zip function
- Programs
- all() and any()



Zip function

Let us understand the zip function with the help of an example



```
lst1=[5,12,15,7]
lst2=[15,8,15,3]

print(list(zip(lst1,lst2)))
```

Output:

```
In [1]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
[(5, 15), (12, 8), (15, 15), (7, 3)]
```

The actual use case of `zip()` is not to zip multiple lists, it can iterate over multiple lists at the same time. Wonder how? Let us see

```
lst1=[5,12,15,7]
lst2=[15,8,15,3]

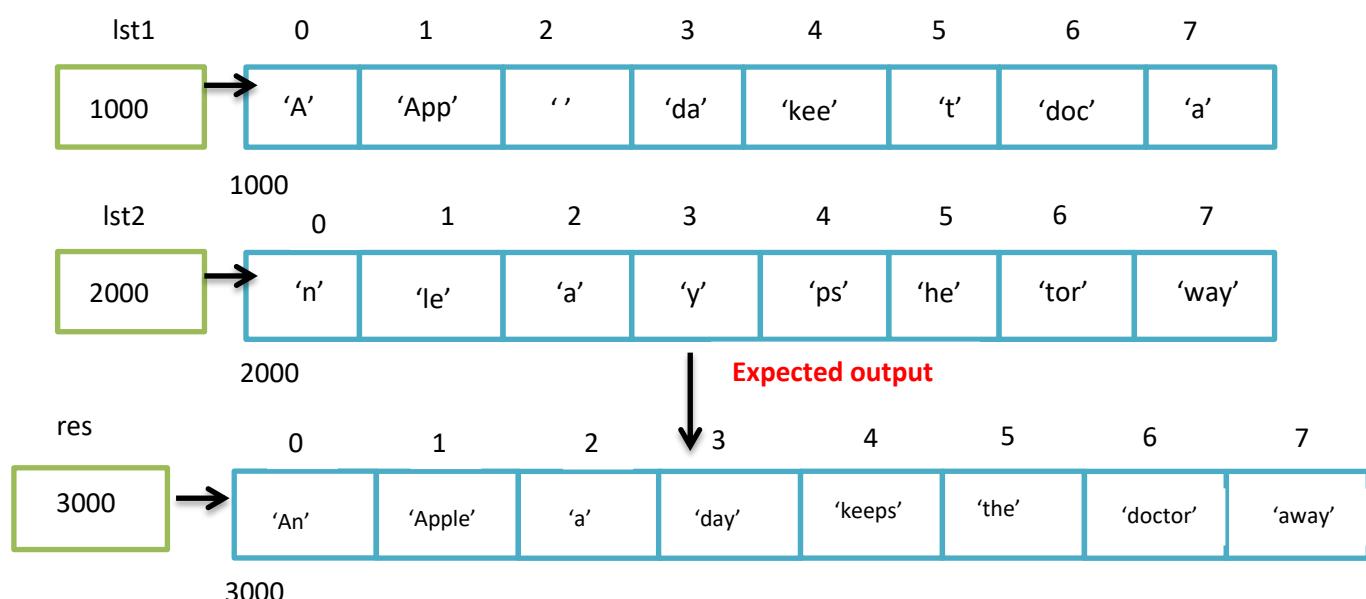
for i,j in zip(lst1,lst2):
    print(i,j)
```

Output:

```
In [2]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
5 15
12 8
15 15
7 3
```



Now our expectation is to join two lists and then form a sentence out of it.



```

lst1=['A','App',' ','da','kee','t','doc','a']
lst2=['n','le','a','y','ps','he','tor','way']
res=[]

for i,j in zip(lst1,lst2):
    res.append(i+j)

print(' '.join(res))

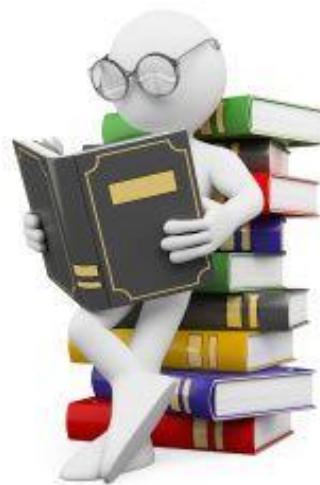
```

Output:

```

In [3]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
An Apple a day keeps the doctor away

```



Let us try to do the same using list comprehension

```

lst1=['A','App',' ','da','kee','t','doc','a']
lst2=['n','le','a','y','ps','he','tor','way']

print(' '.join([i+j for i,j in zip(lst1,lst2)]))

```

Output:

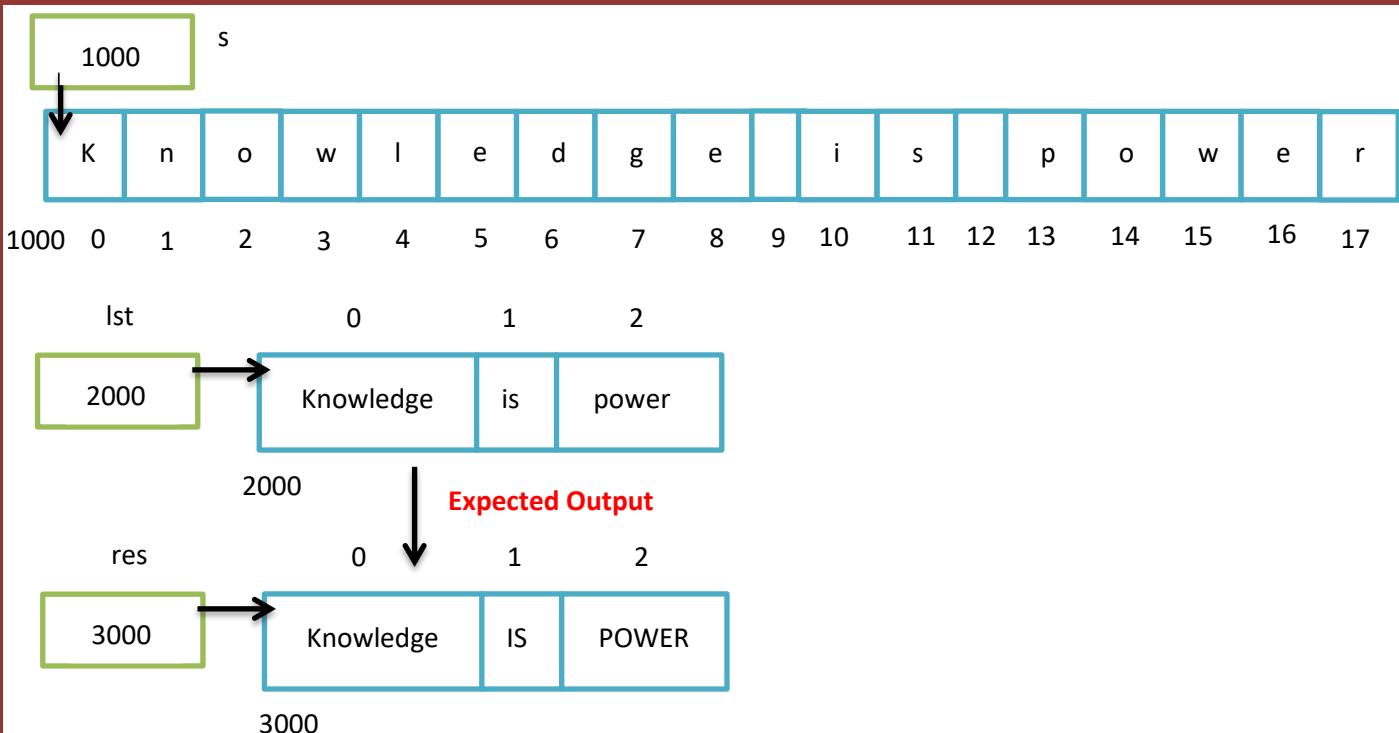
```

In [1]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
An Apple a day keeps the doctor away

```

Programs

Example 1: If the length of word is greater than 5, it should be converted to lowercase and if it is less than or equal to then it should be converted uppercase.



```
s=input("Enter a string:\n")
lst=s.split()
res=[]

for i in range(len(lst)):
    if len(lst[i])>5:
        res.append(lst[i].lower())
    else:
        res.append(lst[i].upper())

print(' '.join(res))
```

Output:

```
In [2]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
```

```
Enter a string:
knowledge is power
knowledge IS POWER
```



Can we reduce the size of code? Definitely we can by making use of list comprehension as shown in the below example

```

s=input("Enter a string:\n")
lst=s.split()

print(' '.join([lst[i].lower() if len(lst[i])>5 else lst[i].upper()
    for i in range(len(lst)) ]))

```

Output:

```

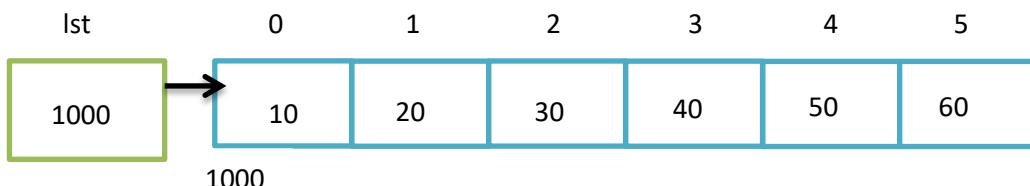
In [3]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')

```

Enter a string:
knowledge is power
knowledge IS POWER

all() and any()

Let us understand what these functions do with an example where we want to check if the elements are positive or not



```

lst=[10,20,30,40,50,60]

print(all([i>0 for i in lst]))
print(any([i>0 for i in lst]))

```

Output:

```

In [1]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
True
True

```

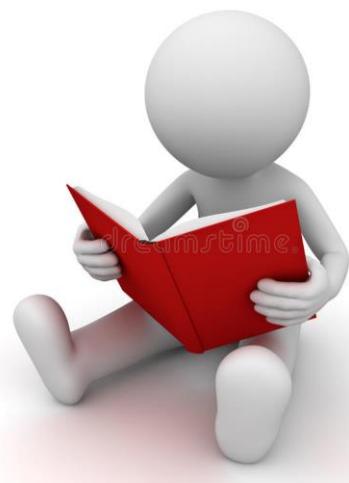
Let us try putting some negative number and then test the output

```
lst=[10,20,30,-40,50,60]

print(all([i>0 for i in lst]))
print(any([i>0 for i in lst]))
```

Output:

```
In [2]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
False
True
```



all() will check the condition for all the elements present whereas
any() will check if at least one element satisfies the condition.

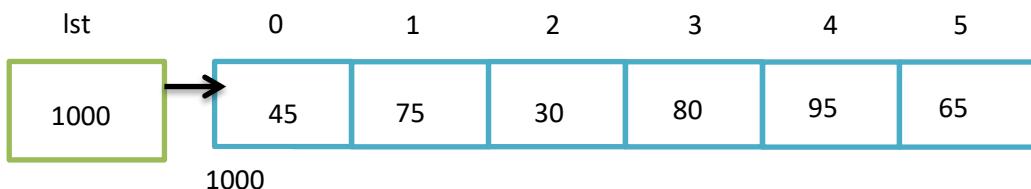
```
lst=[-10,-20,-30,-40,-50,-60]

print(all([i>0 for i in lst]))
print(any([i>0 for i in lst]))
```

Output:

```
In [3]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
False
False
```

Now coming to real time application, let us take an example where we are checking if a student has failed in any of the object or in other words if he/she has passed in all subjects



```
lst=[45,75,30,80,95,65]  
print(all([i>35 for i in lst]))
```

Output:

```
In [4]: runfile('C:/Users/rooman/OneDrive/  
Desktop/python/test.py', wdir='C:/Users/rooman/  
OneDrive/Desktop/python')  
False
```

Now to the same list let us see if any of the marks scored are above 70

```
lst=[45,75,30,80,95,65]  
print(any([i>70 for i in lst]))
```

Output:

```
In [5]: runfile('C:/Users/rooman/OneDrive/  
Desktop/python/test.py', wdir='C:/Users/rooman/  
OneDrive/Desktop/python')  
True
```



Python Fundamentals

day 33

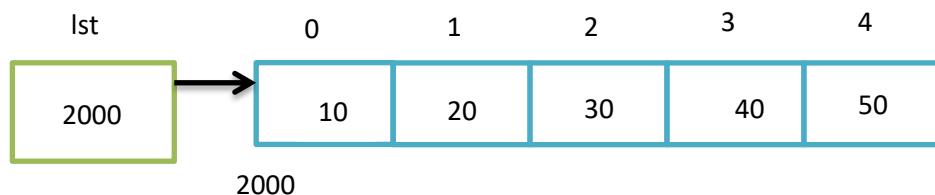
Today's Agenda

- Performance analysis list



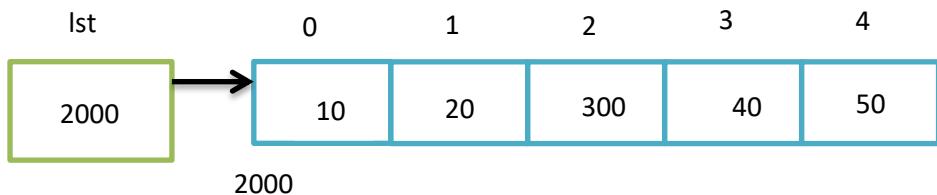
Performance analysis list

Every data structure comes with a certain efficiency of operations on which it performs on the particular system. The software that a programmer code runs in several systems, hence knowing the efficiency of particular data structure is very important for a programmer to make right decisions of when to use and when not to use certain things. To know this we have to perform performance analysis of list. To know what it exactly is, let us consider an example of list and try to **modify its values**



`lst[2]=300`

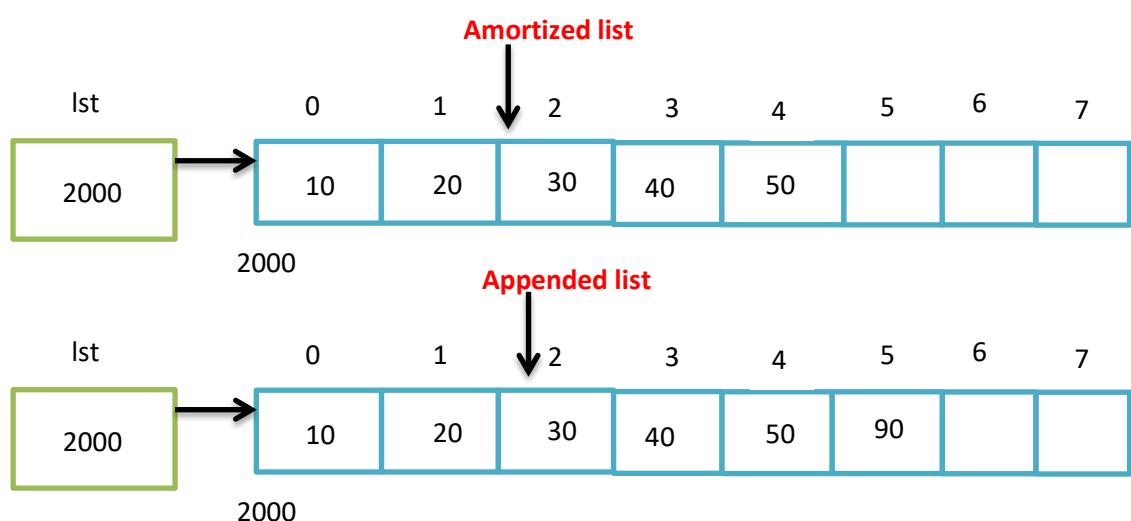
Array arithmetic => $\text{arr}[\text{index}] = \text{base address} + \text{width} * \text{index}$



Using array arithmetic formula we can access any n^{th} element in same time. That means irrespective of the position we can perform the operation in a constant time and index doesn't matter at the time. If we should speak the same thing in terms of time complexity or asymptotic analysis then

Time Complexity → **O(1)** → **1 representing constant time**

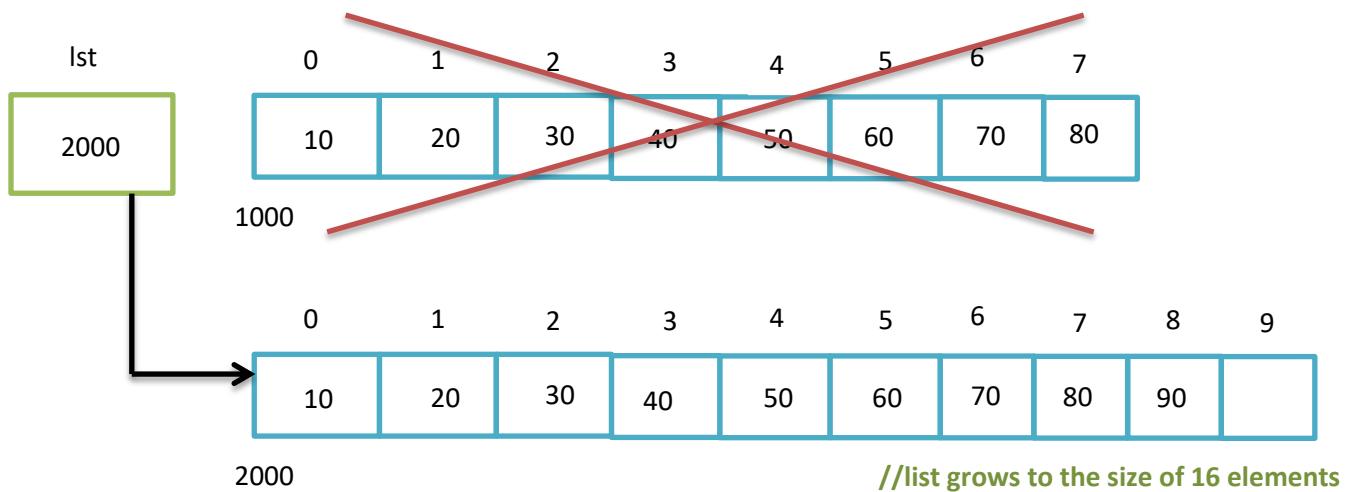
Now let us consider the same list and try to **append an element** and see it's time complexity



Time Complexity → **O(1)** → **1 representing constant time**

It doesn't matter if we are appending a value to a list of length 5 or 50 or 5000. We'll be still appending it to the last element and we know that lists always over allocate the memory. So the time is same for all the elements. Hence time complexity is constant.

But now let us consider that the list is full and to append another value it has to copy all elements and then grow in size and only after that append operation can be done. Does in this case also time complexity remains constant? Let us see



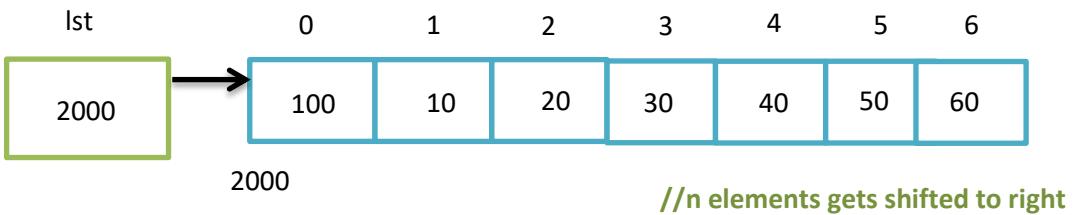
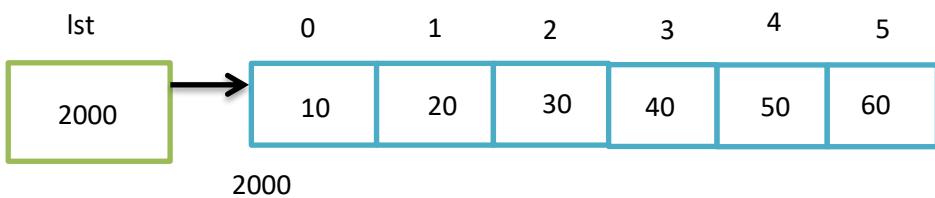
Here time complexity certainly will not be constant because, depending upon the length of the list, new list has to grow in size and then copying operation should be carried only then an element can be appended.

Time Complexity → $O(n)$ → Linear time

Linear time means, as the number of elements increases the time also increases.

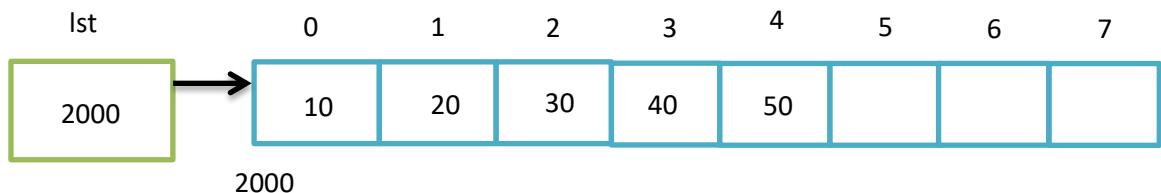
Now let us consider **inserting an element**. Remember that insertion can be done in any position unlike appending at last. So during insertion certain elements must be shifted towards their right and then the element gets inserted (Considering amortized list).

Time complexity is always calculated considering the worst situation which is inserting an element in 0^{th} position where all the elements must be shifted to right and then the new element will be inserted.

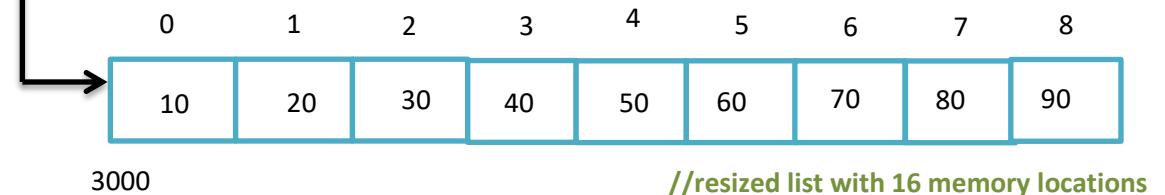
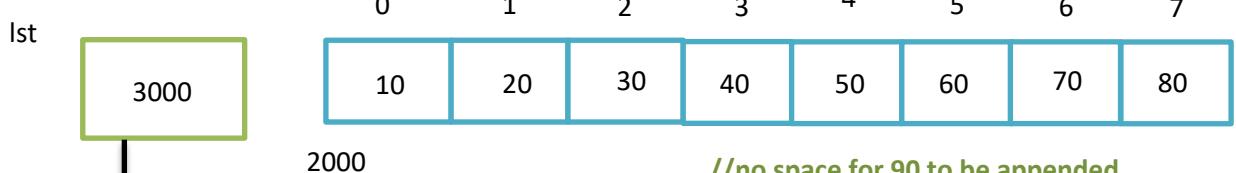


Time Complexity → O(n) → Linear time

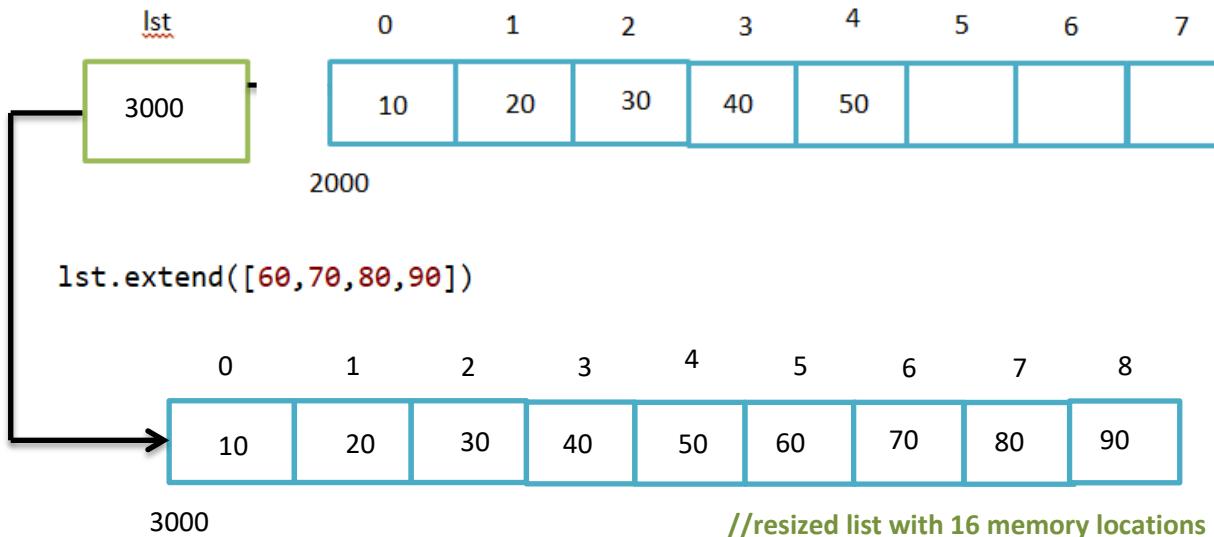
Now let us consider **extending a list**. Which can be done using for loop but let us see what happens if we do that



```
for i in [60,70,80,90]:
    lst.append(i)
```



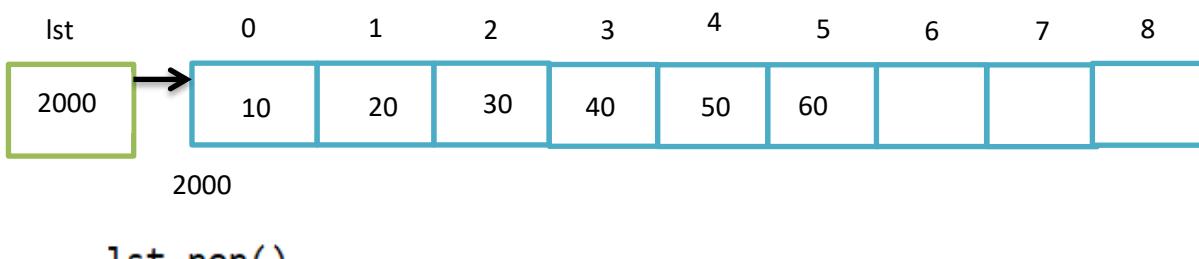
We can see that using for loop is not the efficient way to extend list. There is **extend()** which performs the same operation in efficient way, let us see how

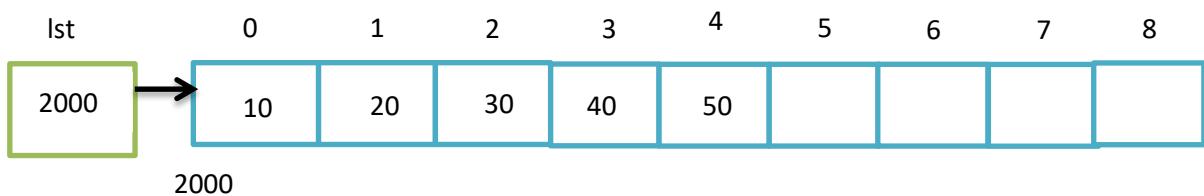


Time Complexity → **O(n²)** → n² represents the no of elements in second list

extend() will check beforehand if all the elements can be placed inside the list, if not the list gets resized and then the operations is carried further. Whereas in previous situation for loop was not checking the vacancies beforehand and only after realisation of no space the new grown list was created and then the elements were copied.

Now next let us consider **popping an element**. In first case let us not pass anything in **pop()** which will result in popping of last element.

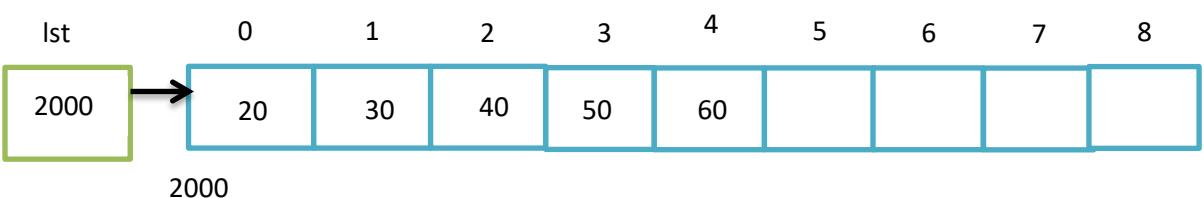




Irrespective of the length of list `pop()` will delete the last element if not mentioned otherwise. Therefore time complexity will remain constant in this case.

Time Complexity → O(1) → 1 represents constant time

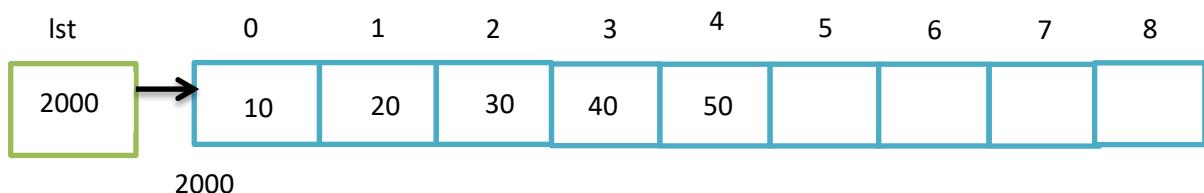
Now what if we pass the index value of the element to be popped? How does it change the time complexity? Let us see, considering the worst case that is, popping the 0th element



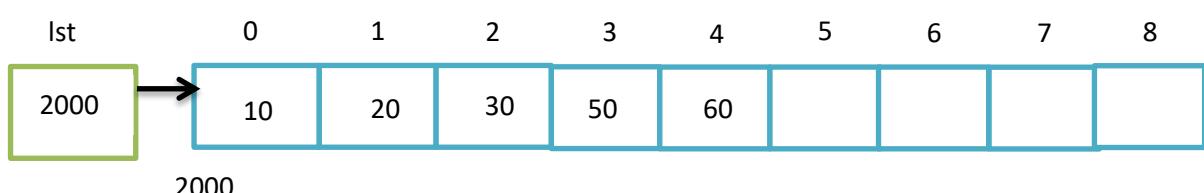
Time Complexity → O(n) → Linear time

All the elements must left shift which is n-1 elements.

Now let us try **removing an element** from the list



`lst.remove(40)`



Time Complexity → O(n) → Linear time

40 should be first searched by iterating on each element and then removed. So complexity in time increases as increase in elements. But what if the element was not present in list? Let us see

```
lst=[10,20,30,40,50]
```

```
lst.remove(99)  
print(lst)
```

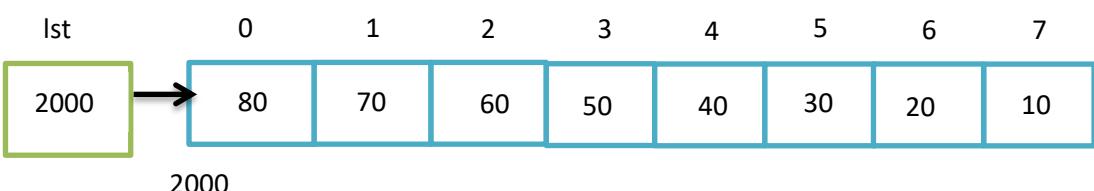
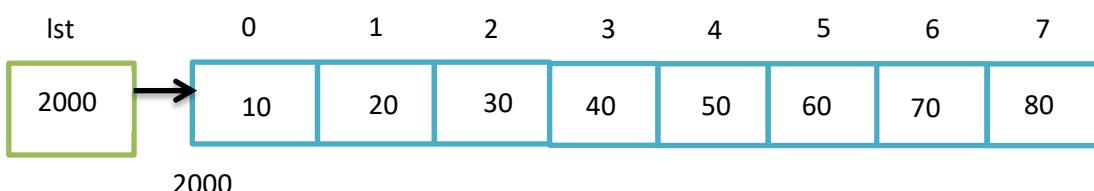
Output:

```
File "C:/Users/rooman/OneDrive/Desktop/  
python/test.py", line 3, in <module>  
    lst.remove(99)  
  
ValueError: list.remove(x): x not in list
```



Certainly we will get an error saying element is not in the list. But to throw this error it has to iterate over n elements and then give a message saying element not in list. So the time complexity is indeed linear in this case as well.

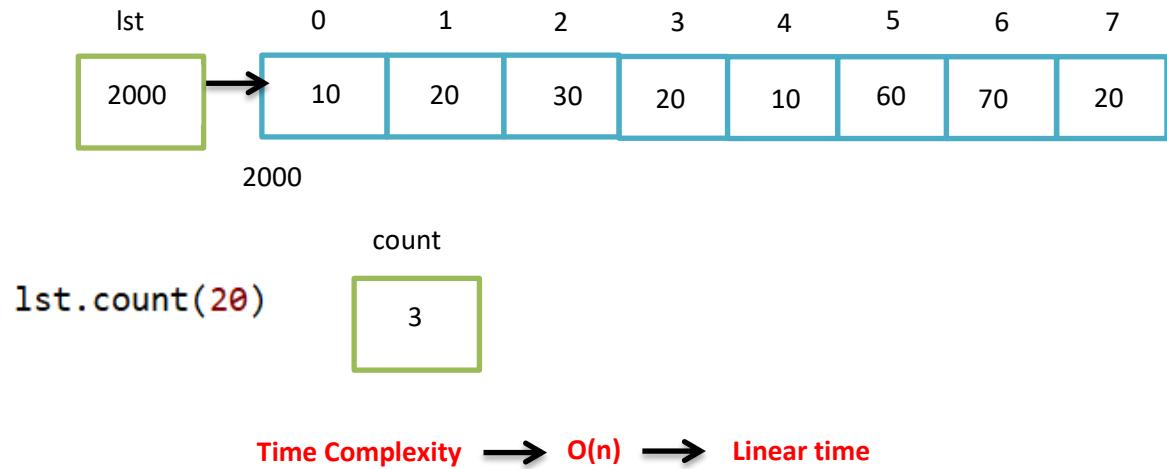
Now next let us try **reversing a list**.



Time Complexity → O(n) → Linear time

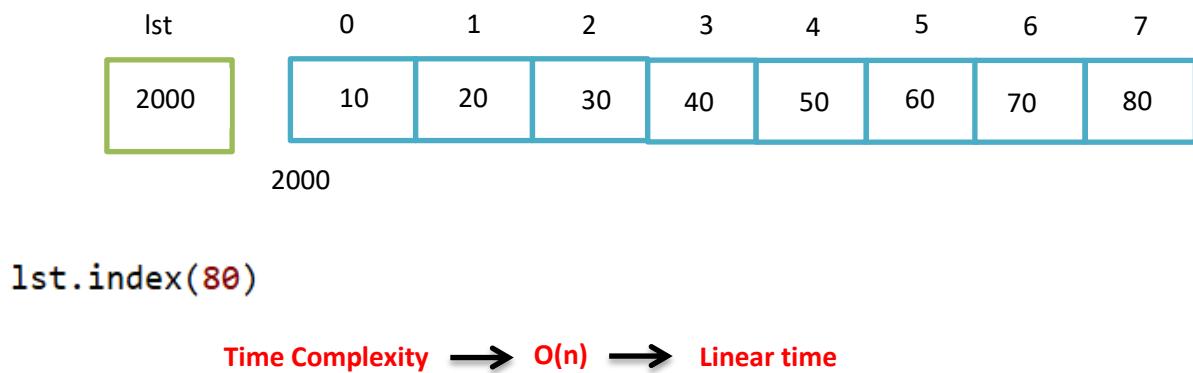
While reversing a list $n/2$ swap operations happen. But that doesn't imply $n/2$ time complexity because as elements increase time complexity will increase which says it is linear in nature.

Next let us have a look at **count of an element**.



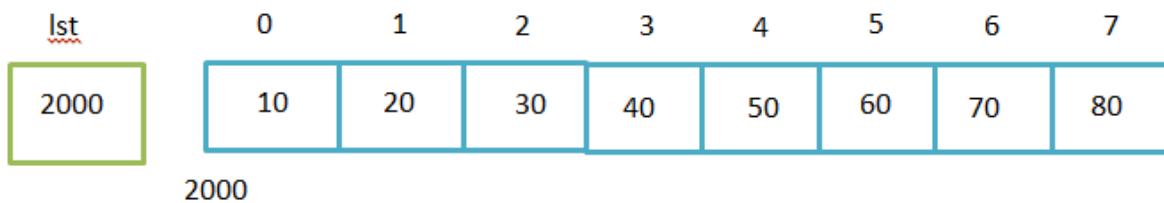
To know the count of an element we certainly have to iterate over each element. Thereby the time complexity will be linear in nature.

Next let us try to find the **index of the element** passed.



Certainly to find the index of the element we need to first search for the element by iterating on the list n times for worst situation. Thereby time complexity will be linear.

Let us now try to **get an element** from the list



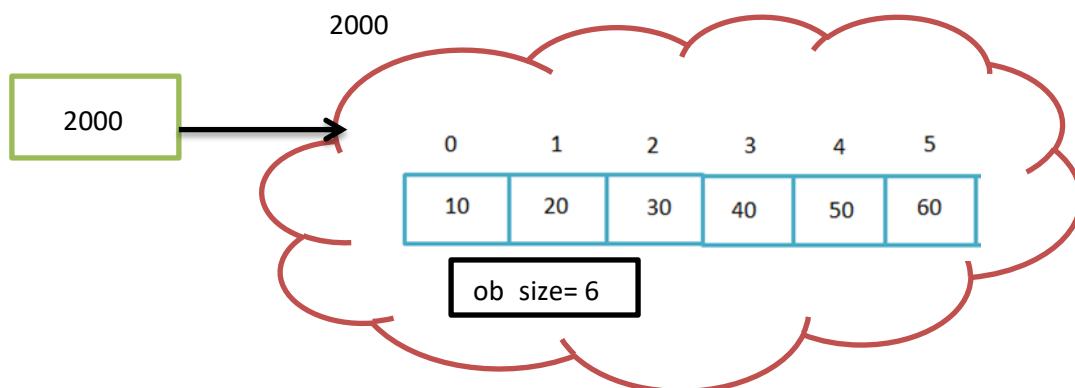
`item=lst[4]`

Array arithmetic => $\text{arr}[\text{index}] = \text{base address} + \text{width} * \text{index}$

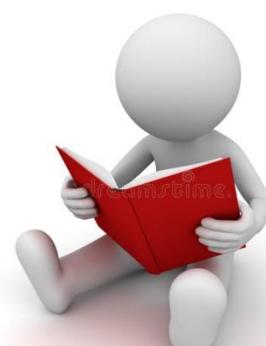
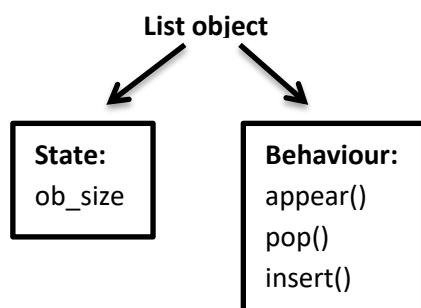
Irrespective of the length of the list, getting an element will take constant time as we are mentioning the index of the element we want to get.

Time Complexity $\rightarrow O(1) \rightarrow$ Constant time

Next let us try with **`len()` function**



Time Complexity $\rightarrow O(1) \rightarrow$ Constant time



List is an object which has certain states in which a variable called as `ob_size` stores the length of list. So irrespective of the size of list the length can be accessed easily by the help of this variable and hence the time complexity is constant.



Python fundamentals

day 34

Today's Agenda

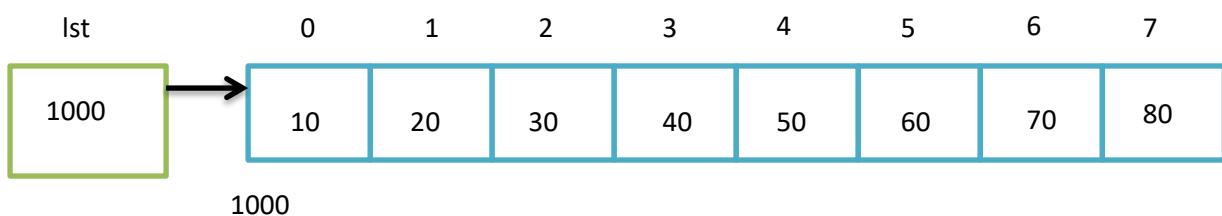
- Time complexity of operators
- Tuples



Time complexity of operators

Let us now begin with finding the time complexity of operators like `in`, `==`, `!=`, `<`, `<=`, `>`, `>=`, `+`, `*` in list.

Let us consider an example for `in` operator where we are trying to find a value in list.

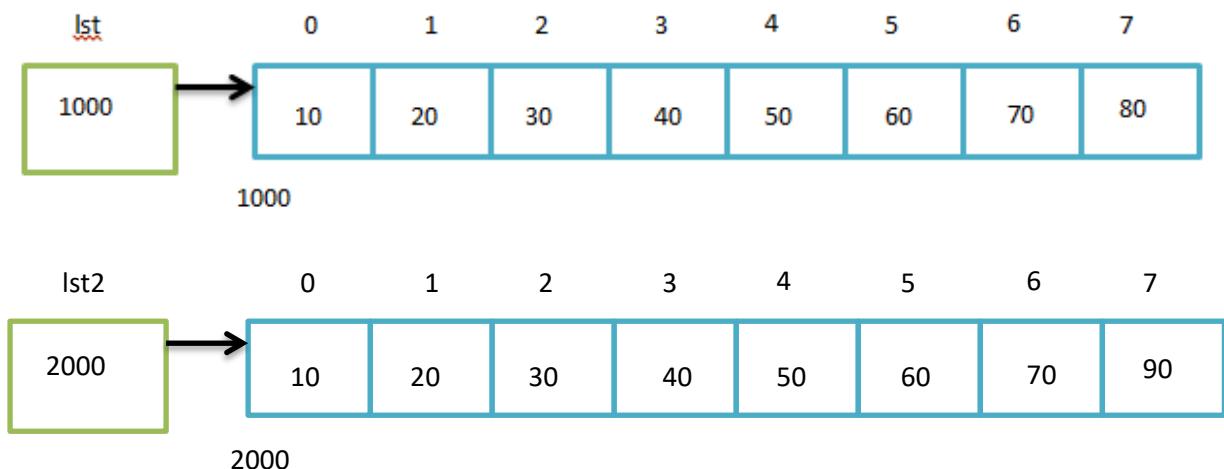


99 in 1st

Time Complexity → O(n) → Linear time

When we use `in` operator, in worst situation every element is checked.

Now let us see the comparison operators : **$==$, \neq , $<$, $<=$, $>$, \geq**

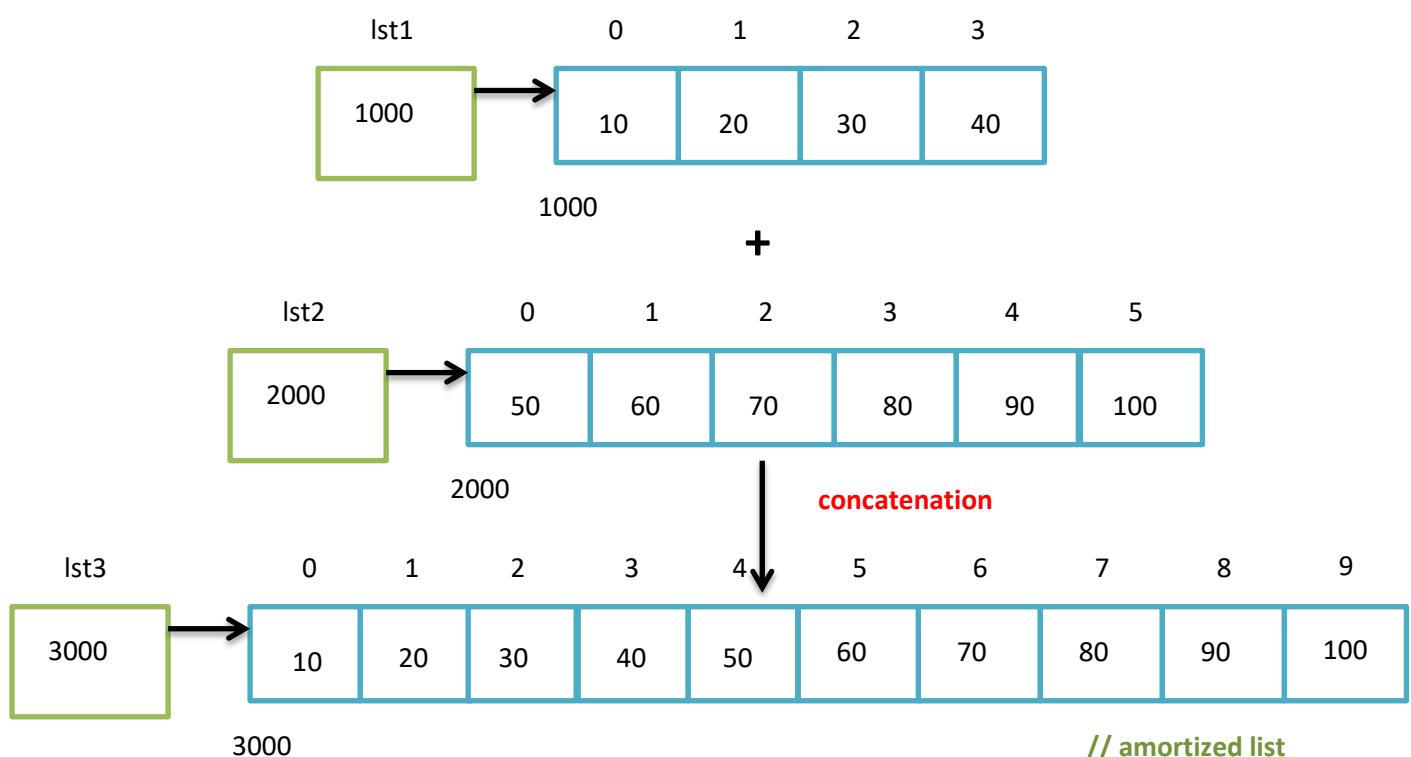


`lst == lst1`

Time Complexity $\rightarrow O(n) \rightarrow$ Linear time

Amongst all operators let us now consider `$=$` operator, which has to check each value to know if two lists are equal or not. Same case with other operators as well.

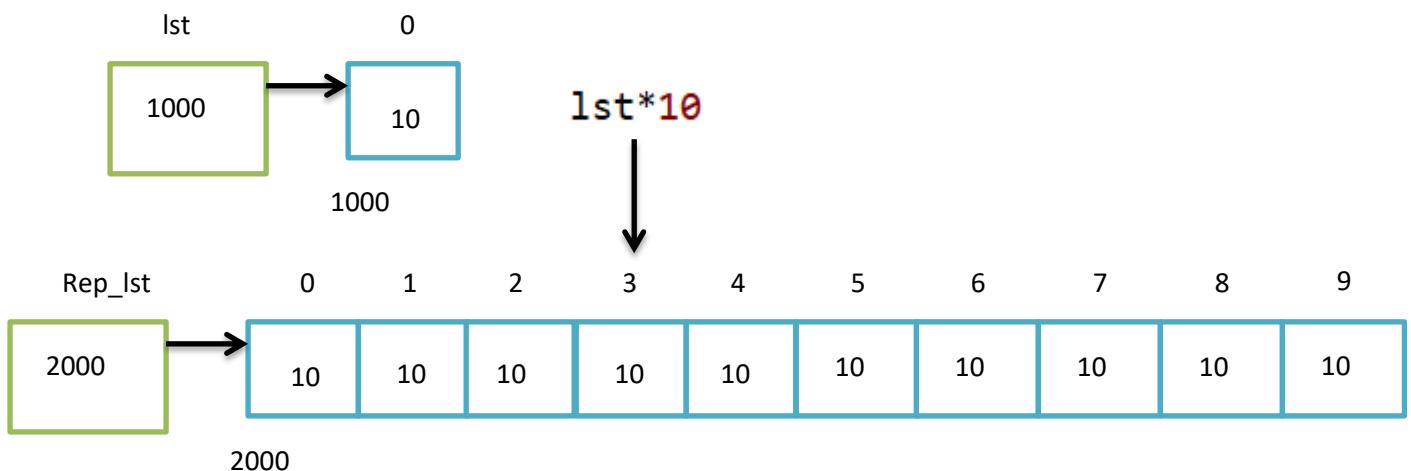
Now let's see **`+` operator (Concatenation)**



Time Complexity → $O(n_1+n_2)$ → Linear time

As each elements copy has to be made before concatenation, the time complexity will be linear.

Last but not the least let us see * operator (Replication)



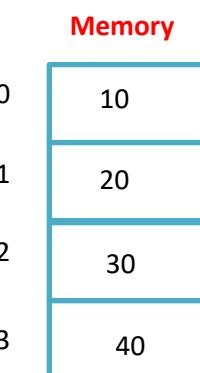
Time Complexity → $O(c*n)$ → Linear time

C is number of times n has to be repeated, which increases linearly.

Tuple

Tuples are similar to lists, but tuples are immutable in nature and internally they make use of static arrays and not dynamic arrays. Unlike in lists here static array means they cannot grow or shrink in size. Let us understand through an example

```
t=(10,20,30,40)  
print(t)  
#trying to add an element  
t.append(99)  
print(t)
```



Output:

```
File "C:/Users/rooman/OneDrive/Desktop/
python/test.py", line 4, in <module>
    t.append(99)

AttributeError: 'tuple' object has no attribute
'append'
```

In tuple we don't have option of appending or removing elements. Let us see different ways of creating a tuple

```
tup=()
print(tup)
```

Output:

```
In [2]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
()
```

Next way of creating a tuple is

```
tup=tuple() #using in-built function
print(tup)
```

Output:

```
In [3]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
()
```

Storing homogeneous data

```
tup=(10,20,30,40,50)
print(tup)
```



Output:

```
In [4]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
(10, 20, 30, 40, 50)
```

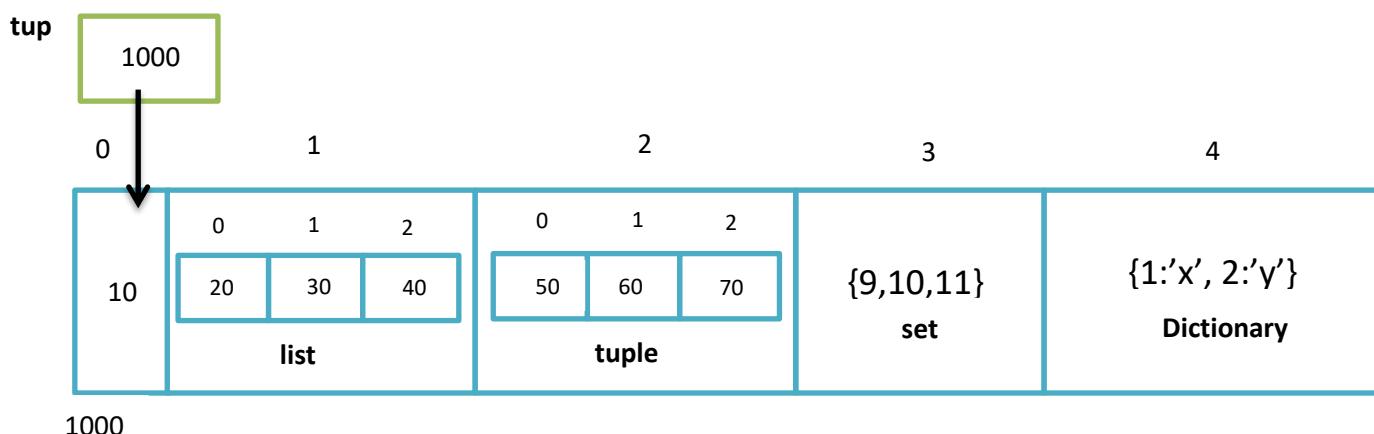
Can we store heterogeneous data as well? Let us give a try

```
tup=(10,20.5,True,1+3j,'Python')
print(tup)
```

Output:

```
In [5]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
(10, 20.5, True, (1+3j), 'Python')
```

Great!! Now let us see if just like lists can tuple also store a tuple, list, set, dictionary in it



```
tup=(10,[20,30,40],(50,60,70),{9,10,11},{1:'x',2:'y'})
print(tup)
```

Output:

```
In [6]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
(10, [20, 30, 40], (50, 60, 70), {9, 10, 11},
{1: 'x', 2: 'y'})
```

Now let us see does tuple support the features that list supported

```
tup=(10,20.5,True,1+3j,'Python')
print(tup)
#tup.append(99)
#tup.insert(2,99)
#tup.extend((60,70,80))
```

Output:



```
File "C:/Users/rooman/OneDrive/Desktop/
python/test.py", line 5, in <module>
    tup.extend((60,70,80))
```

```
AttributeError: 'tuple' object has no attribute
'rerror'
```

We will get similar error saying that those attributes doesn't exist in tuple data structure.

Let us see other modifications considering same example

```
tup=(10,20.5,True,1+3j,'Python')
print(tup)
#tup[2]=False
#tup.pop()
#tup.pop(2)
#tup.remove(1+3j)
```

Output:



```
File "C:/Users/rooman/OneDrive/Desktop/
python/test.py", line 6, in <module>
    tup.remove(1+3j)
```

```
AttributeError: 'tuple' object has no attribute
'remove'
```

We certainly cannot change anything inside a tuple. But we definitely can access individual elements or portion of elements. let us see in below example

```
tup=(10,20.5,True,1+3j,'Python')
print(tup)
print(tup[2])
print(tup[1:4])
```

Output:

```
In [9]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
(10, 20.5, True, (1+3j), 'Python')
True
(20.5, True, (1+3j))
```

Now let us try to create a tuple of single element and see what happens

```
tup=(10)
print(tup)
print(type(tup))
```

Output:

```
In [10]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
10
<class 'int'>
```



We see that type of object is not tuple instead it is int. Whenever we try to store a single element or singleton tuple there's a certain way to do it, let us see how to do it

```
tup=(10,) #creating a tuple with single element
print(tup)
print(type(tup))
```

Output:

```
In [11]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
(10,)
<class 'tuple'>
```

Let us see another interesting thing about tuple

```
tup=10,20,30 #packing  
print(tup)  
print(type(tup))
```

Output:

```
In [12]: runfile('C:/Users/rooman/OneDrive/  
Desktop/python/test.py', wdir='C:/Users/rooman/  
OneDrive/Desktop/python')  
(10, 20, 30)  
<class 'tuple'>
```

Even without () the elements are treated as tuple because they are packed inside a tuple.



Python fundamentals

day 35

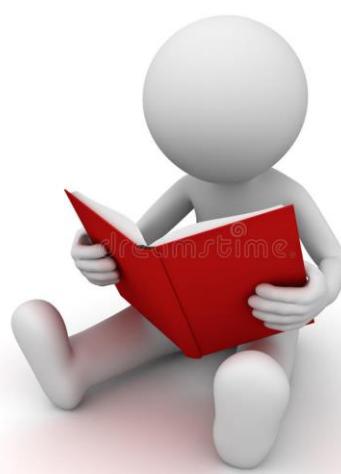
Today's Agenda

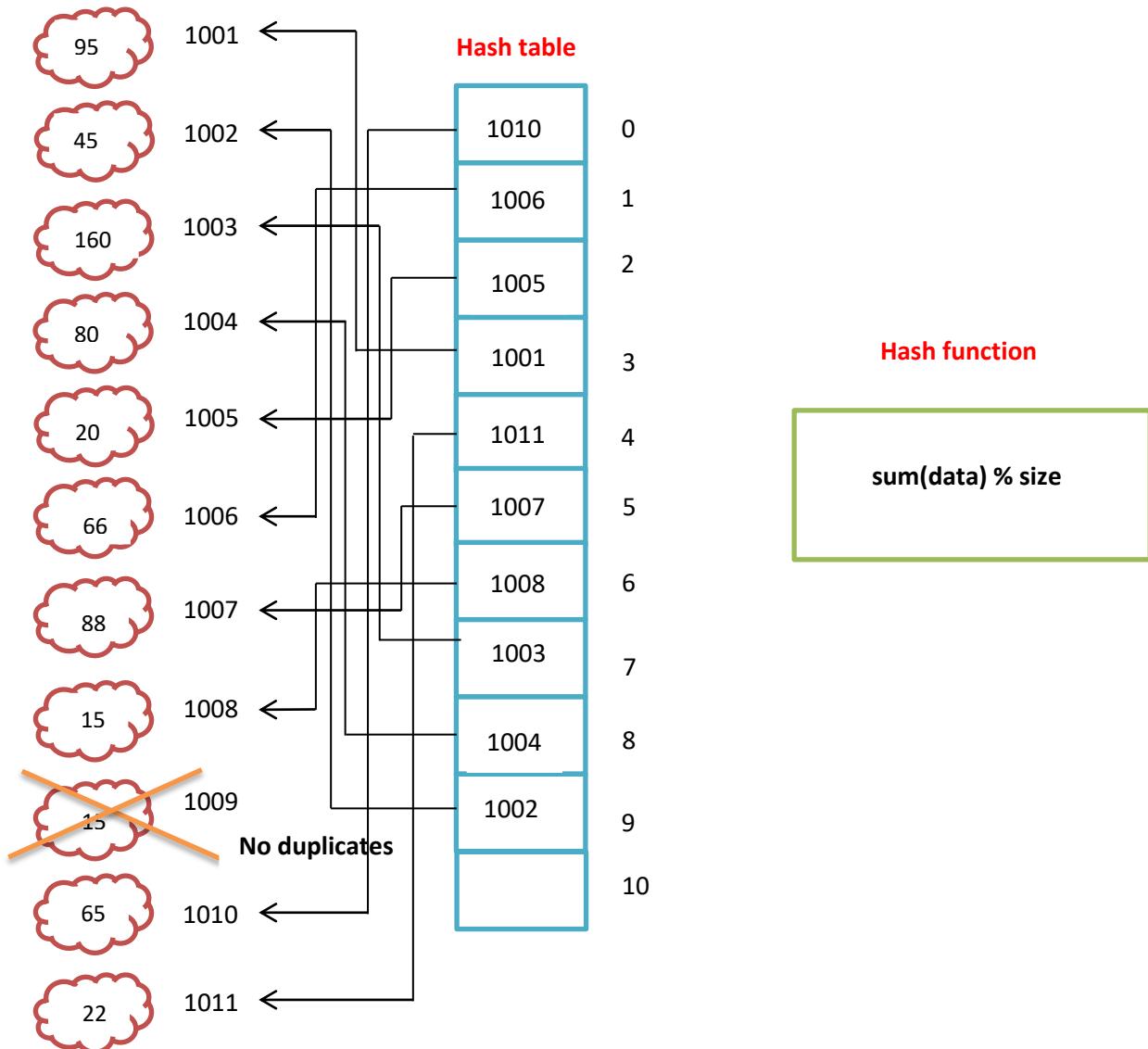
- Set
- Separate chaining



Set

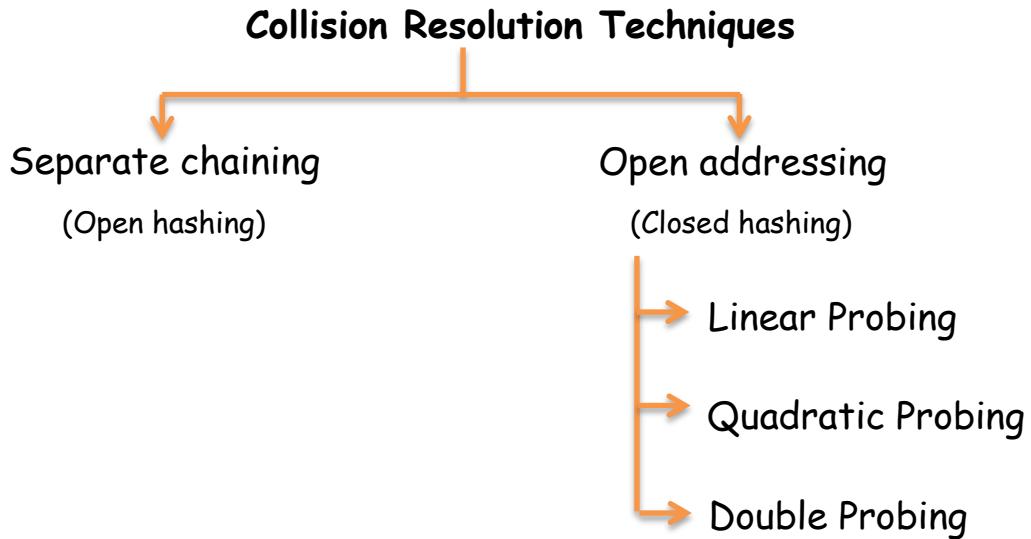
We know that set is a collection which is unordered and un-indexed. In Python sets are written with curly brackets. To understand sets in better way and to know why they are unordered, we should first know the internal implementation of it. So let us consider an example and get to know it





Sets use a methodology called **hashing** to store data. There are two components associated with hashing hash functions and has table.

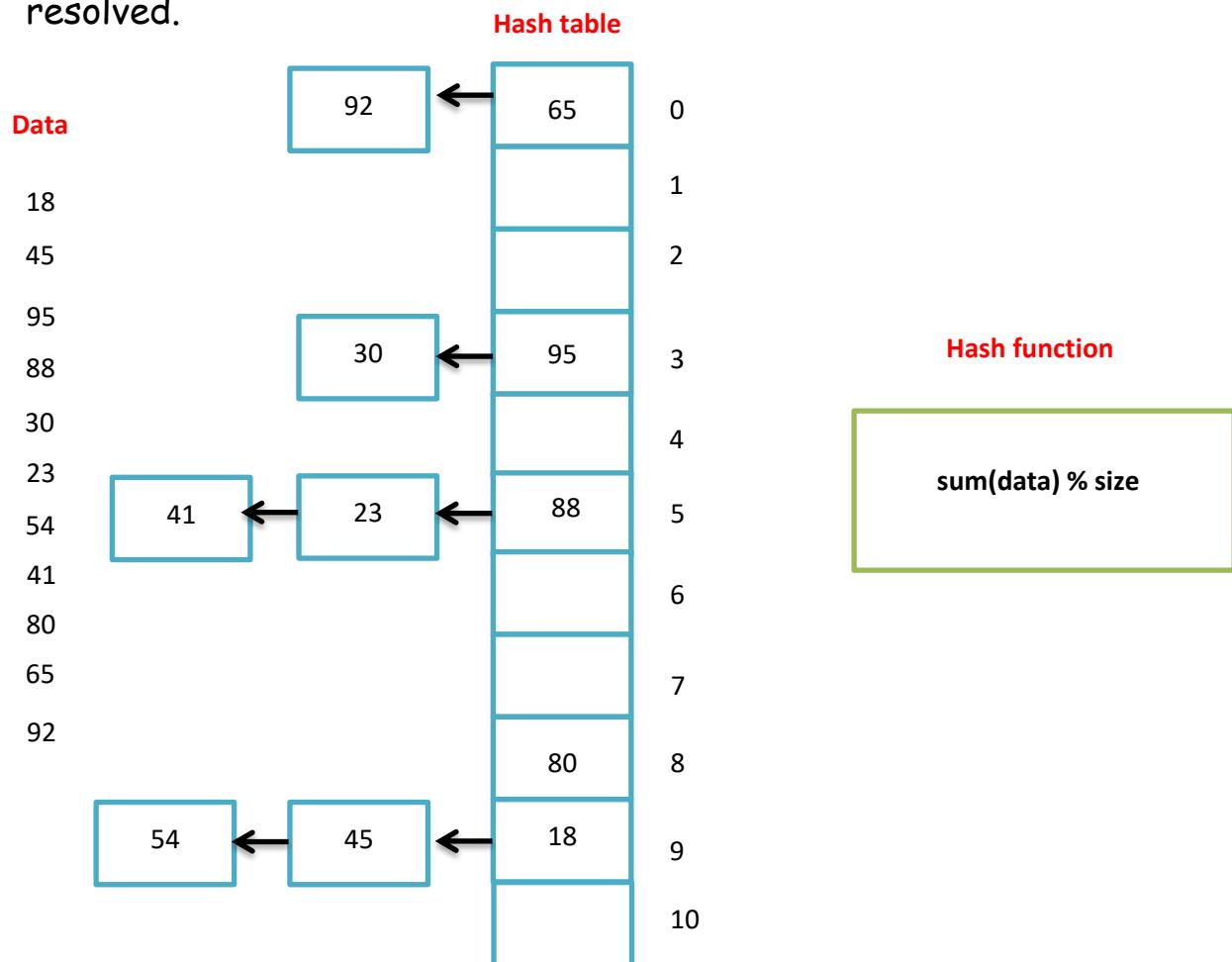
Now what if we have another data in the set as 92, its sum is 11 (9+2) and $11 \% 11$ is 0. But at 0th position we already have a value stored. At this time collision occurs. Is there a way to resolve this issue? Certainly it can be solved by using **collision resolution techniques**



Amongst all, let us focus on separate chaining.

Separate chaining

Let us consider the following example and see how collision is resolved.



Whenever collision happens a separate block is chained to the same position and in the new block the value is stored. This is known as separate chaining.

So far we have seen that set uses hashing to store data, set is unordered data, and set will not allow duplicate entries to store.

The next feature of set is that, inside a set we cannot store mutable datatypes. Is there any particular reason for this? Let us see

To understand this we should first know hashable data / immutable data.

Hashable / Immutable data

- ❖ Hash key is calculated using all the data.
- ❖ Hash key must be unique and should not change.

Why can't a set be stored in a set

Let us see what happens if we store mutable data inside a set.

We know that, to generate the hash key all the data must be used. Now what if we store a mutable data like list and get a hash key, but later on what if we want to append an element or remove an element certainly the data is going to change and so would the hash key. And we cannot have two hash keys for the same object. Hence the entire hashing algorithm would be faulty.



Let us see an example with code

```
s={95,45,160,80,20,66,88,15,15,65,22}
print(s)
```



Output:

```
In [1]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
{160, 65, 66, 45, 15, 80, 20, 22, 88, 95}
```

We can see that the output is unordered, duplicates have been removed. And note that the hash function computer uses is different and complicated than the one which is explained above for your understanding.

Now let us see if we can store immutable data in set

```
s={10,3.14,True,1+3j,(40,50,60),'Python'}
print(s)
```

Output:

```
In [2]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
{True, 3.14, 10, (1+3j), (40, 50, 60),
'Python'}
```

Great! We can store immutable data. What about mutable data? Let us see

```
s=[[10,20,30],{66,77,88},{1:'A',2:'B'}]
print(s)
```

Output:

```
File "C:/Users/rooman/OneDrive/Desktop/
python/test.py", line 1, in <module>
s=[[10,20,30],{66,77,88},{1:'A',2:'B'}]

TypeError: unhashable type: 'list'
```

We certainly cannot store mutable datatypes. If you are wondering can we access the hash key generated by hash function; Let us see in the next example

```
print(hash(99))
print(hash(99.9))
```

Output:

```
In [4]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
99
2075258708292337763
```

```
print(hash([10,20,30]))
```

Output:

```
File "C:/Users/rooman/OneDrive/Desktop/
python/test.py", line 1, in <module>
    print(hash([10,20,30]))
TypeError: unhashable type: 'list'
```

`hash()` definitely will not accept mutable datatypes, it will only accept hashable/imutable datatypes.

Let us see how set is a mutable datatype

```
s={5,78,92,77,105,3,67}
print(s)
s.add(999)
print(s)
s.remove(999)
print(s)
s.pop() #will pop random element
print(s)
```



Output:

```
In [6]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
{3, 67, 5, 105, 77, 78, 92}
{3, 67, 5, 999, 105, 77, 78, 92}
{3, 67, 5, 105, 77, 78, 92}
{67, 5, 105, 77, 78, 92}
```

Let us see some more features of set

```
s={5,78,92,77,105,3,67}
print(s)
#s.remove(111) Will give an error as the element is not present in set
s.discard(111)
print(s)
```

Output:

```
In [7]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
{3, 67, 5, 105, 77, 78, 92}
{3, 67, 5, 105, 77, 78, 92}
```

discard() will check if the element is present in the set, if yes it removes it and if not it just discards and doesn't give any error.

```
s={5,78,92,77,105,3,67}
print(s)
s.update({33,44,56})
print(s)
```

Output:

```
In [8]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
{3, 67, 5, 105, 77, 78, 92}
{33, 3, 67, 5, 105, 44, 77, 78, 56, 92}
```

The set is updated, definitely in unordered way.

Python Fundamentals

day 36

Today's Agenda

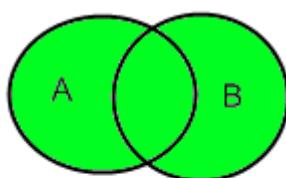
- Set operations
- Subset
- Programs on sets



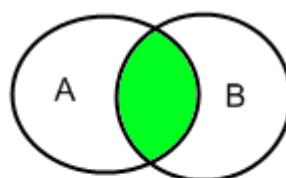
Set Operations

Operations performed on two sets at the same time are called as set operations. Set operations include set union, set intersection, set difference, complement of set and Cartesian product.

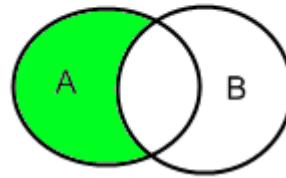
Union of A and B



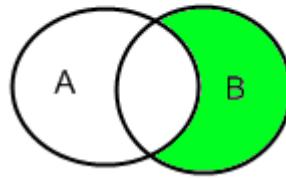
Intersection of A and B



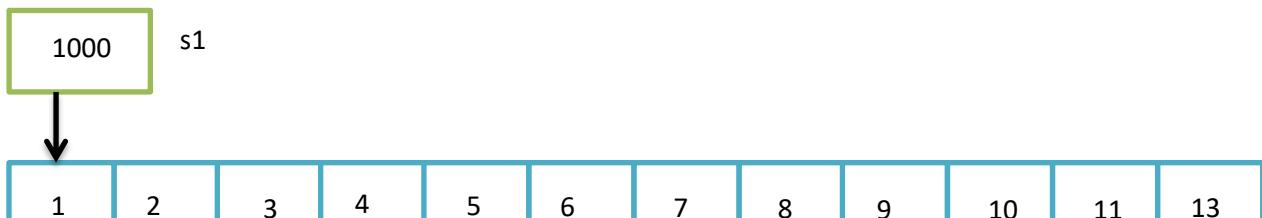
Difference A minus B



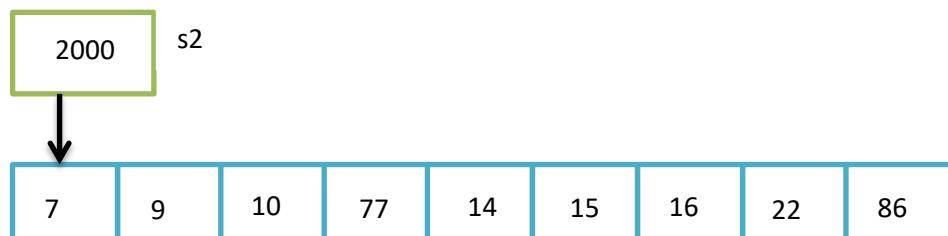
Difference B minus A



Let us consider an example and see the operations



1000



2000

Union of s1 and s2 is the set of elements present in s1, in s2 or in both s1 and s2.

```
s1={1,2,3,4,5,6,7,8,9,10,11,13}
s2={7,9,10,77,14,15,16,22,86}
s3=s1|s2 #union using operator
print(s1)
print(s2)
print(s3)
```

Output:

```
In [13]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13}
{7, 9, 10, 77, 14, 15, 16, 22, 86}
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13, 77, 14,
15, 16, 22, 86}
```



```

s1={1,2,3,4,5,6,7,8,9,10,11,13}
s2={7,9,10,77,14,15,16,22,86}
s3=s1.union(s2) #union using built-in function
print(s1)
print(s2)
print(s3)

```

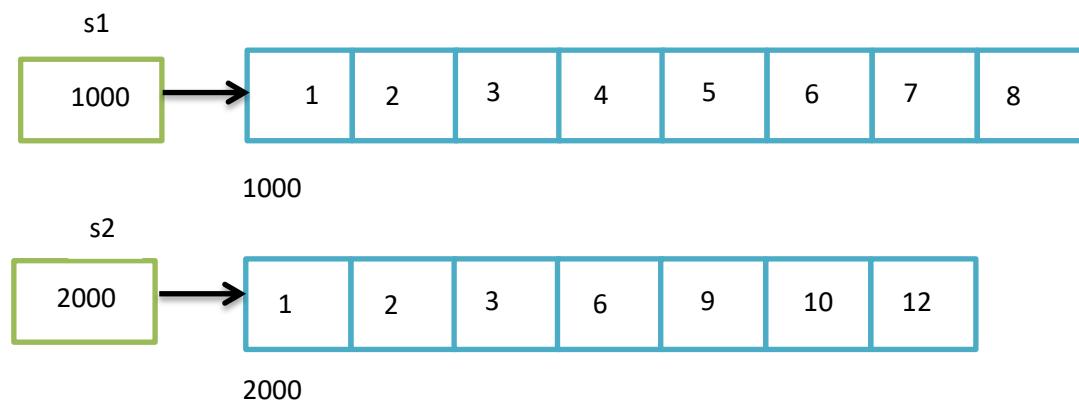
Output:

```

In [14]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13}
{7, 9, 10, 77, 14, 15, 16, 22, 86}
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13, 77, 14,
15, 16, 22, 86}

```

Next let us see the intersection of two sets. Intersection of two given sets s1 and s2 is a set which consists of all the elements which are common to both s1 and s2.



```

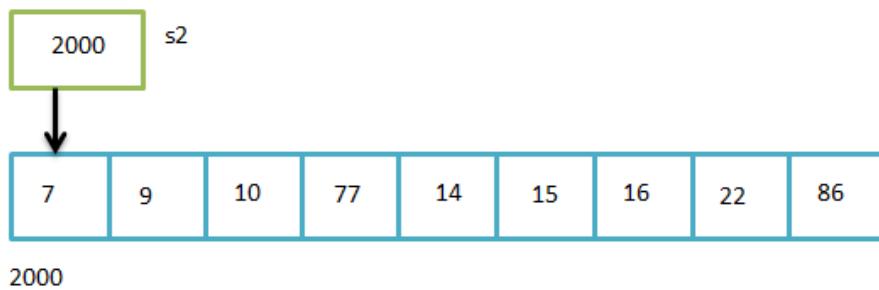
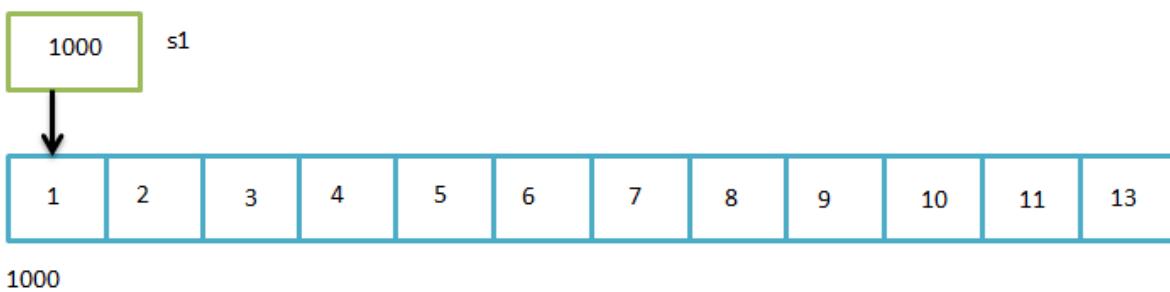
s1={1,2,3,4,5,6,7,8}
s2={1,2,3,6,9,10,12}
s3=s1&s2 #intersection using operator
s4=s1.intersection(s2) #intersection using built-in function
print(s1)
print(s2)
print(s3)
print(s4)

```

Output:

```
In [15]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
{1, 2, 3, 4, 5, 6, 7, 8}
{1, 2, 3, 6, 9, 10, 12}
{1, 2, 3, 6}
{1, 2, 3, 6}
```

Next operation is the difference of two sets. Difference of two sets considering $s1 - s2$ will return the elements in $s1$ after removing the elements of $s2$ in $s1$.



```
s1={1,2,3,4,5,6,7,8,9,10,11,13}
s2={7,9,10,77,14,1,5,16,22,86}
s3=s1-s2 #difference using operator
s4=s2-s1
s5=s1.difference(s2) #difference using built-in function
s6=s2.difference(s1)
print(s1)
print(s2)
print(s3)
print(s4)
print(s5)
print(s6)
```

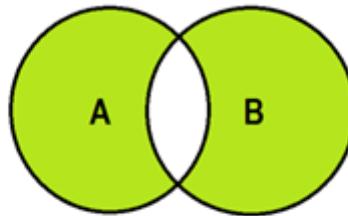
Output:

```
In [16]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13}
{1, 5, 7, 9, 10, 77, 14, 16, 22, 86}
{2, 3, 4, 6, 8, 11, 13}
{77, 14, 16, 86, 22}
{2, 3, 4, 6, 8, 11, 13}
{77, 14, 16, 86, 22}
```



Note: $s_1 - s_2$ is not same as $s_2 - s_1$.

Next let us see the symmetric difference of two sets. The symmetric difference of two sets s_1 and s_2 is the set of elements that are in either A or B , but not in their intersection.



`A.symmetric_difference(B) or A^B`

```
s1={1,2,3,4,5,6,7,8,9,10,11,13}
s2={7,9,10,77,14,1,5,16,22,86}
s3=s1^s2 #symmetric difference using operator
s4=s1.symmetric_difference(s2) #using built-in function
print(s1)
print(s2)
print(s3)
print(s4)
```

Output:

```
In [17]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13}
{1, 5, 7, 9, 10, 77, 14, 16, 22, 86}
{2, 3, 4, 6, 8, 11, 77, 14, 13, 16, 22, 86}
{2, 3, 4, 6, 8, 11, 77, 14, 13, 16, 22, 86}
```

Now let us see some more built-in functions of set operations:

intersection_update(): Will update the first set as the intersection result of the two sets.

```
s1={1,2,3,4,5,6,7,8,9,10,11,13}
s2={7,9,10,77,14,15,16,22,86}
print(s1)
s1.intersection_update(s2)
print(s1)
```

Output:

```
In [20]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13}
{9, 10, 7}
```

difference_update(): Will update the first set as the difference result of the two sets.

```
s1={1,2,3,4,5,6,7,8,9,10,11,13}
s2={7,9,10,77,14,15,16,22,86}
print(s1)
s1.difference_update(s2)
print(s1)
```

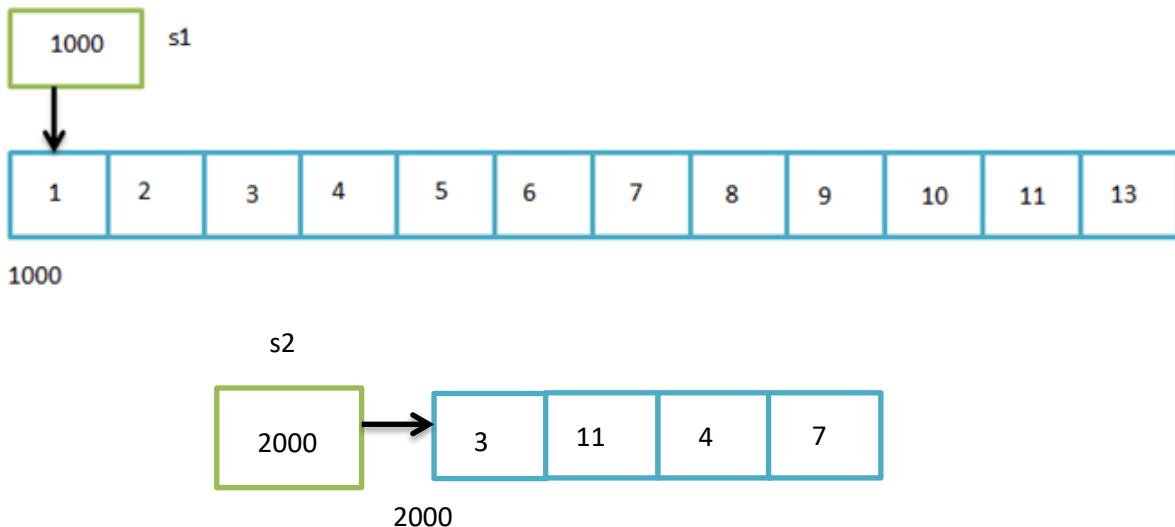
Output:

```
In [21]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/r
OneDrive/Desktop/python')
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13}
{1, 2, 3, 4, 5, 6, 8, 11, 13}
```



Subset, Superset and disjoint set

A set of which all the elements are contained in another set is called as a subset.



```
s1={1,2,3,4,5,6,7,8,9,10,11,13}
s2={3,11,4,7}
print(s2<=s1)
print(s2.issubset(s1))
```

Output:

```
In [1]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
True
True
```



A set(s1) is said to be superset if all the elements of s2 are in s1.

```
s1={1,2,3,4,5,6,7,8,9,10,11,13}
s2={3,11,4,7}
print(s1>=s2)
print(s1.issuperset(s2))
```

Output:

```
In [2]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
True
True
```

Two sets are said to be disjoint when their intersection is null, or in other words when the two sets have nothing in common.

```
s1={1,2,3,4,5,6,7,8,9,10,11,13}
s2={18,19,20,66}
print(s1.isdisjoint(s2))
```

Output:

```
In [3]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
True
```

Programs on set

Example 1: Take the input from the user and remove all the duplicates from it.

```
lst=input().split()
print(lst)
print(set(lst))
```

Output:

```
In [4]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')

java python c java
['java', 'python', 'c', 'java']
{'c', 'java', 'python'}
```



Example 2: Write a program to print the number of duplicate elements in the list.

```
lst=list(map(int,input().split()))
s=set(lst)
print(len(lst)-len(s))
```

Output:

```
In [5]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
10 22 14 22 8 16 14 18 66
2
```

Example 3: Given 3 set of roll numbers of students who play hockey, football and cricket. Print the roll numbers according to the below conditions.

- a) Who play any game
- b) Who play all 3 games
- c) Who play only hockey
- d) Who play either football or cricket but not both

```
h={1,9,12,7,14}
f={6,9,8,10,5,11,12,13,15}
c={2,4,9,3,5,13}
print(h|f|c) #play any - answer(a)
print(h&f&c) #play hockey,football and cricket - answer(b)
print(h-(f|c)) #only hockey - answer(c)
print(f^c) #either football or hockey - answer(d)
```

Output:

```
In [6]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15}
{9}
{1, 14, 7}
{2, 3, 4, 6, 8, 10, 11, 12, 15}
```

Python Fundamentals

day 37

Today's Agenda

- Set comprehension
- Programs
- Frozen set



Set Comprehension

Similar to list comprehension python also has set comprehension. Let us understand with an example

```
#traditional way
s={5,4,9,8,7,2}
#res={} - treated as dictionary, not set
res=set()

for i in s:
    res.add(i**2)

print(s)
print(res)
```

Output:

```
In [1]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
{2, 4, 5, 7, 8, 9}
{64, 4, 16, 49, 81, 25}
```

```
#using set comprehension
s={5,4,9,8,7,2}
res={i**2 for i in s}

print(s)
print(res)
```

Output:

```
In [3]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
{2, 4, 5, 7, 8, 9}
{64, 4, 16, 49, 81, 25}
```

Now our expectation is to find the square of
only even numbers

```
#traditional way
s={5,4,9,8,7,2}
res=set()

for i in s:
    if i%2==0:
        res.add(i**2)

print(s)
print(res)
```



Output:

```
In [4]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
{2, 4, 5, 7, 8, 9}
{16, 64, 4}
```

```
#using set comprehension
s={5,4,9,8,7,2}
res={i**2 for i in s if i%2==0 }

print(s)
print(res)
```

Output:

```
In [5]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
{2, 4, 5, 7, 8, 9}
{16, 64, 4}
```

Let us see if the syntax changes after including `else` condition

```
#traditional way
s={5,4,9,8,7,2}
res=set()

for i in s:
    if i%2==0:
        res.add(i**2)
    else:
        res.add(i+i)

print(s)
print(res)
```



Output:

```
In [6]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
{2, 4, 5, 7, 8, 9}
{64, 4, 10, 14, 16, 18}
```

```
#Set comprehension
s={5,4,9,8,7,2}
res={i**2 if i%2==0 else i+i for i in s}

print(s)
print(res)
```

Output:

```
In [7]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
{2, 4, 5, 7, 8, 9}
{64, 4, 10, 14, 16, 18}
```

Programs

Example 1: Check is a string is pangram.

```
s='The Quick Brown Fox Jumps Over a Lazy Dog'  
s=s.upper()  
c=set()  
  
for i in s:  
    if ord(i)>=65 and ord(i)<=90:  
        c.add(i)  
  
if len(c) ==26:  
    print(s,'is a pangram')  
else:  
    print(s,'is not a pangram')
```



Output:

```
In [8]: runfile('C:/Users/rooman/OneDrive/  
Desktop/python/test.py', wdir='C:/Users/rooman/  
OneDrive/Desktop/python')  
THE QUICK BROWN FOX JUMPS OVER A LAZY DOG is a  
pangram
```

```
s='An Apple a Day Keeps The Doctor Away'  
s=s.upper()  
c=set()  
  
for i in s:  
    if ord(i)>=65 and ord(i)<=90:  
        c.add(i)  
  
if len(c) ==26:  
    print(s,'is a pangram')  
else:  
    print(s,'is not a pangram')
```



Output:

```
In [9]: runfile('C:/Users/rooman/OneDrive/  
Desktop/python/test.py', wdir='C:/Users/rooman/  
OneDrive/Desktop/python')  
AN APPLE A DAY KEEPS THE DOCTOR AWAY is not a  
pangram
```

Can the above code be more efficient and smaller? Certainly using set comprehension it can be achieved

```
s='An Apple a Day Keeps The Doctor Away'  
s=s.upper()  
  
if len({i for i in s if ord(i)>=65 and ord(i)<=90}) ==26:  
    print(s,'is a pangram')  
else:  
    print(s,'is not a pangram')
```

Output:

```
In [10]: runfile('C:/Users/rooman/OneDrive/  
Desktop/python/test.py', wdir='C:/Users/rooman/  
OneDrive/Desktop/python')  
AN APPLE A DAY KEEPS THE DOCTOR AWAY is not a  
pangram
```

Example 2: Check if the string is heterogram.

```
s='The Big Dwarf Only Jumps!'.upper()  
l=[]  
  
for i in s:  
    if ord(i)>=65 and ord(i)<=90:  
        l.append(i)  
  
c=set(l)  
  
if len(l)==len(c):  
    print(s,'is heterogram')  
else:  
    print(s,'is not heterogram')
```

Output:

```
In [14]: runfile('C:/Users/rooman/OneDrive/  
Desktop/python/test.py', wdir='C:/Users/rooman/  
OneDrive/Desktop/python')  
THE BIG DWARF ONLY JUMPS! is heterogram
```

Can the code be written using list comprehension? Definitely it can



```
s='The Big Dwarf Only Jumps!'.upper()
l=[i for i in s if ord(i)>=65 and ord(i)<=90]

c=set(1)

if len(l)==len(c):
    print(s,'is heterogram')
else:
    print(s,'is not heterogram')
```

Output:

```
In [15]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
THE BIG DWARF ONLY JUMPS! is heterogram
```

```
s='The Big Dwarf Only Jumpsss!'.upper()
l=[i for i in s if ord(i)>=65 and ord(i)<=90]

c=set(1)

if len(l)==len(c):
    print(s,'is heterogram')
else:
    print(s,'is not heterogram')
```

Output:

```
In [16]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
THE BIG DWARF ONLY JUMPSSS! is not heterogram
```



Frozen set

We know that set is mutable. But what if we don't want it to be mutable? Is there a way? Definitely. A set can be made immutable by wrapping it in a built-in function called as `frozenset()` which will disable the operations involving modification.

```
s=frozenset({32,67,45,16,21})
s1={45,89,32,15}
#s.pop - will throw error
#s.remove(45) - will throw error
#s.add(99) - will throw error
#print(s.intersection_update(s1)) - will throw error
print(s|s1)
print(s&s1)
```

Output:

```
In [17]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
frozenset({32, 67, 45, 15, 16, 21, 89})
frozenset({32, 45})
```

The following operations can't be performed as they modify the existing set

`add()`, `update()`, `remove()`, `discard()`, `pop()`, `difference_update()`,
`intersection_update()`, `symmetric_difference_update()`

The following operation are allowed to be performed on frozenset as they don't modify the existing set

`union()`, `intersection()`, `difference()`,
`symmetric_difference()`,
`isdisjoint()`, `issubset()`, `issuperset()`



Python Fundamentals

day 38

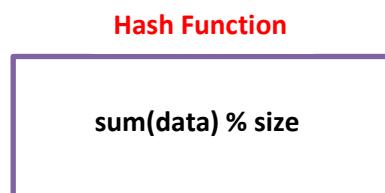
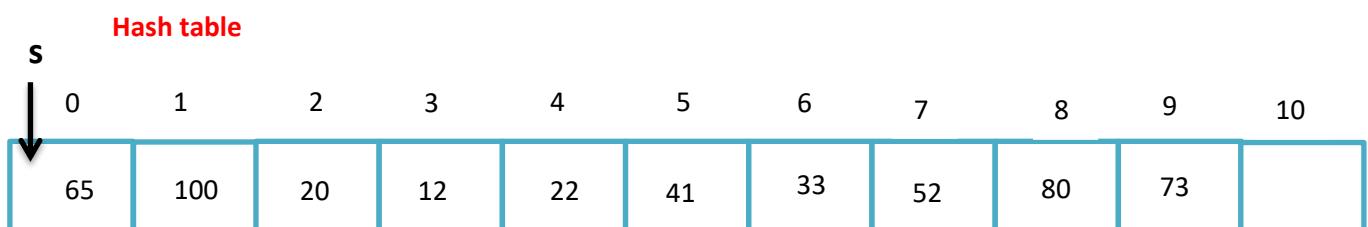
Today's Agenda

- Performance analysis of sets
- Dictionary



Performance analysis of sets

Let us see the performance analysis of several operations in sets.
Consider the following example with respective hash table and hash function



To **add** an element:

`s.add(45)` is an operation of constant time irrespective of the size of the set.

Time complexity: $O(1) \rightarrow$ Constant Time

To **remove** an element:

`s.remove(45)` is an operation of constant time irrespective of the size of the set.

Time complexity: $O(1) \rightarrow$ Constant Time

Popping of an element:

`s.pop()` is also an operation with constant time irrespective of the size of the set.

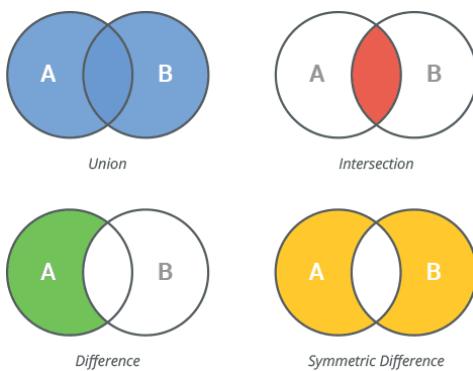
Time complexity: $O(1) \rightarrow$ Constant Time

Search of an element:

Unlike in lists, search operation `33 in s` is very efficient in sets as each element is stored based on its hash key. Therefore even the search operation is constant time.

Time complexity: $O(1) \rightarrow$ Constant Time

Now let us see the time complexity of set operations like union, intersection, difference and symmetric difference etc



Union of two sets:

When union of two sets is taken then each and every element is considered from both the sets. So with increase in number of elements in the set, time complexity increases.

Time complexity: $O(n_1+n_2)$ → Linear Time

Intersection of sets:

When we are finding the intersection or the elements common in both the sets, we need to check with all the elements present. So here also the time complexity is linear.

Time complexity: $O(n_1+n_2)$ → Linear Time

Difference of sets:

Even while finding the difference all the elements are considered so here as well the time complexity increases as increase in number of elements.

Time complexity: $O(n_1+n_2)$ → Linear Time

Symmetric difference of sets:

Similar to the difference of sets, time complexity in symmetric difference also increases as increase in number of elements.

Time complexity: $O(n_1+n_2)$ → Linear Time

Checking if s_2 is subset of s_1 :

To check if s_2 is subset of s_1 , we need to see if all the elements of s_2 are present in s_1 . So time complexity depends on number of elements in s_2 .

Time complexity: $O(n_2)$ → Linear Time

Checking if s1 is **superset** of s2:

To check if s1 is superset of s2, again we need to see if all the elements of s2 are present in s1. So time complexity depends on number of elements in s2.

Time complexity: $O(n^2)$ → Linear Time

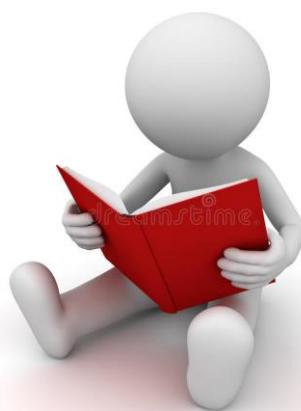
Checking if s1 and s2 are **disjoint** sets:

To check if both s1 and s2 are disjoint sets, we need to check each and every element and see they are present in both the sets or not. So the time complexity increases as increase in number of elements.

Time complexity: $O(n_1+n_2)$ → Linear Time

Dictionary

Earlier we saw how elements are stored in lists, tuple and set. We saw the efficiency of operations like search. Where lists take linear time to search for an element set takes constant time. But we can store duplicate elements in list, we can access each element using its index, but this is not possible in sets. So is there any data structure which can store duplicate elements, mutable, immutable data with index based access and more importantly without compromising the time efficiency, certainly to achieve all of the above we need to use dictionaries.



Let us see how the elements are stored in dictionary

$d \rightarrow \{2:45, 1:92, 16:64, 22:56, 18:72, 26:88\}$



The key difference in storing an element in dictionary is, instead of the values passing through hash function they key is passed and based on the hash key generated the key and its associated value gets stored. So all the restrictions are now imposed on the key and not to the values.

Let us see different ways of keys passed in dictionary

```
d={2:45,1:92,16:64,22:56,18:72,26:88}  
print(d[2])  
print(26 in d)
```

Output:

```
In [3]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
45
True
```

We can also pass string as a key

```
d={'two':45,1:92,16:64,22:56,18:72,26:88}
print(d['two'])
print(26 in d)
```

Output:

```
In [4]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
45
True
```

We can pass all immutable data structures as keys. Only the mutable data structures cannot be passed as keys.

Now let us see how to add elements to dictionary

```
d={1:'C',2:'Java',3:'Python'}
print(d)
d[4]='C++' #adding single element
print(d)
d.update({5:'C#',6:'VisualBasics'}) #adding multiple elements
print(d)
d.update(seven='Javascript',eight='PHP')
print(d)
d[2]='Python' #passing duplicate
print(d)
```

We are trying to add a single element, add multiple elements in two different ways using **update()** and try to modify existing element and pass an existing value.

Output:

```
In [5]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
{1: 'C', 2: 'Java', 3: 'Python'}
{1: 'C', 2: 'Java', 3: 'Python', 4: 'C++'}
{1: 'C', 2: 'Java', 3: 'Python', 4: 'C++', 5: 'C#', 6: 'VisualBasics'}
{1: 'C', 2: 'Java', 3: 'Python', 4: 'C++', 5: 'C#', 6: 'VisualBasics', 'seven': 'Javascript', 'eight': 'PHP'}
{1: 'C', 2: 'Python', 3: 'Python', 4: 'C++', 5: 'C#', 6: 'VisualBasics', 'seven': 'Javascript', 'eight': 'PHP'}
```



Python fundamentals

day 39

Today's Agenda

- Built-in methods in Dictionary
- Storing elements
- Accessing elements in dictionary



Built in methods in dictionary

Let us see some how to modify dictionary using some built-in methods



```
d={1:'p',2:'y',3:'t',4:'h',5:'o',6:'n',7:'a'}
print(d)
d.pop(3) #pops an element with they key value
d.popitem() #pops the last element
del d[4] # deletes the element with key value
print(d)
```

Output:

```
In [2]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
{1: 'p', 2: 'y', 3: 't', 4: 'h', 5: 'o', 6:
'n', 7: 'a'}
{1: 'p', 2: 'y', 5: 'o', 6: 'n'}
```

Let us see some features of `pop()`

```
d={1:'p',2:'y',3:'t',4:'h',5:'o',6:'n',7:'a'}
print(d.pop(3)) #returns the value that is popped
#print(d.pop(99)) will throw keyerror
print(d.pop(99,"Not Found")) #error can be avoided by giving default value
```

Output:

```
In [3]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
t
Not Found
```

```
d={1:'p',2:'y',3:'t',4:'h',5:'o',6:'n',7:'a'}
print(d)
d.clear() #clears the elements from dictionary
print(d)
#print(d.popitem()) throws keyerror
```

Output:

```
In [6]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
{1: 'p', 2: 'y', 3: 't', 4: 'h', 5: 'o', 6:
'n', 7: 'a'}
{}
```

Storing elements

Let us understand by an example where we have stored a string and a list in dictionary

```
d={1:'a',2:[10,20,30]}
print(d[1])
x=d[1]
print(x)
x='b'
print(d[1])
print(x)
```



Output:

```
In [7]: runfile('C:/Users/rooman/OneDrive/  
Desktop/python/test.py', wdir='C:/Users/rooman/  
OneDrive/Desktop/python')  
a  
a  
a  
b
```

Great the output is as expected where the dictionary values remained intact. Let us see what happens to list if we do the same operations

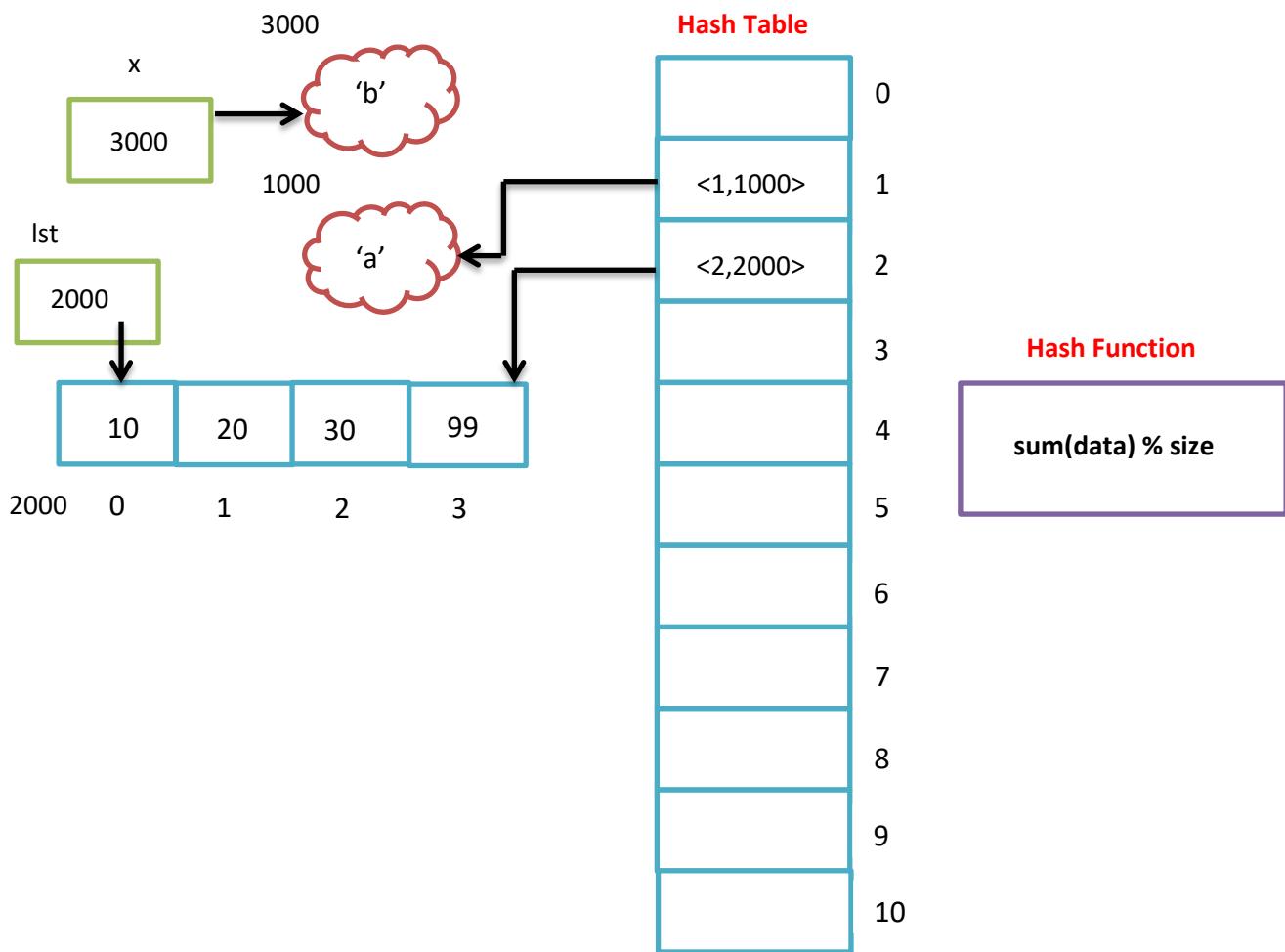
```
d={1: 'a', 2:[10,20,30]}  
print(d[2])  
lst=d[2]  
print(lst)  
lst.append(99)  
print(d[2])  
print(lst)
```

Output:

```
In [8]: runfile('C:/Users/rooman/OneDrive/  
Desktop/python/test.py', wdir='C:/Users/rooman/  
OneDrive/Desktop/python')  
[10, 20, 30]  
[10, 20, 30]  
[10, 20, 30, 99]  
[10, 20, 30, 99]
```

We can see that the output is not as expected. The list value inside dictionary has also changed. Wonder why? Let us see through memory perspective





Accessing elements in dictionary

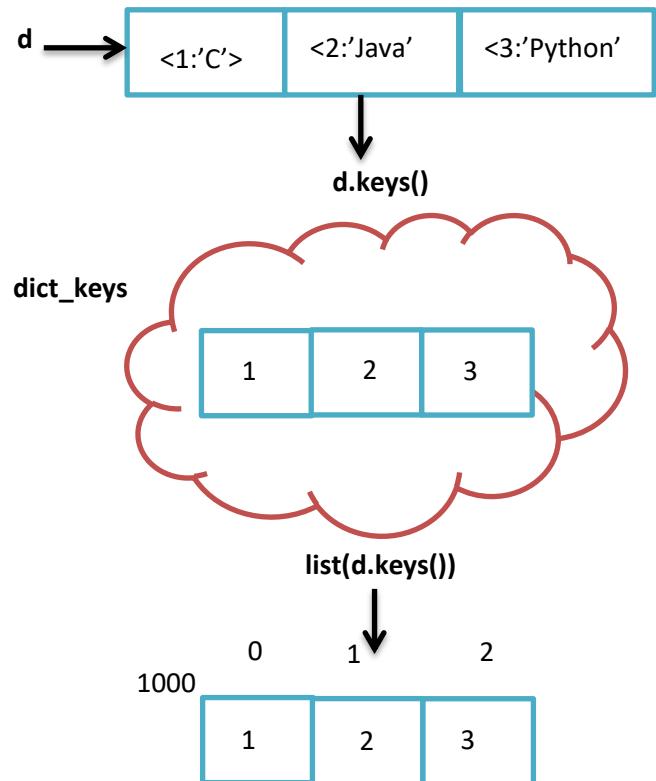
We know there are keys and their values in dictionary. Let us see how to access keys first

```
d={1:'C',2:'Java',3:'Python'}
print(list(d.keys()))
for i in d.keys():
    print(i,d[i])
```

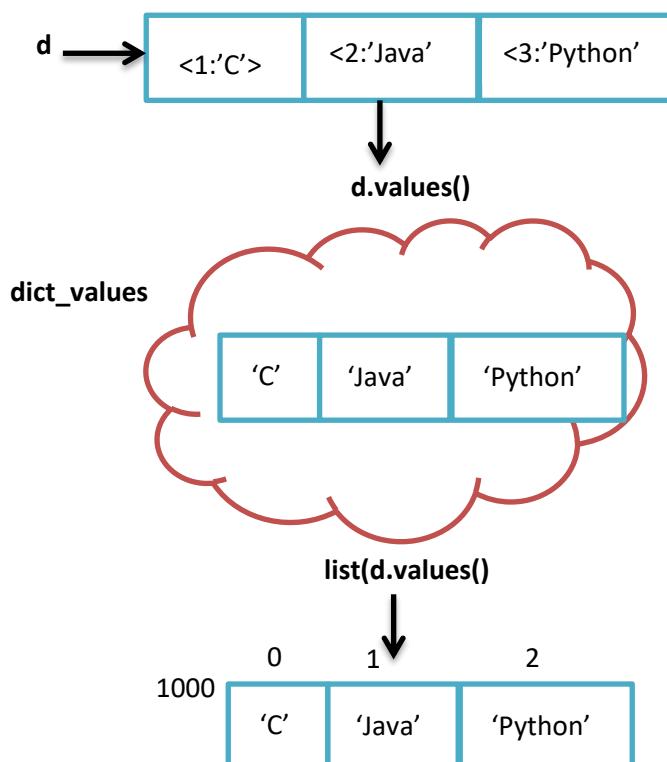
Output:

```
In [10]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
[1, 2, 3]
1 C
2 Java
3 Python
```





Next let us see how to access only the values



```

d={1:'C',2:'Java',3:'Python'}
print(list(d.values()))
for i in d.values():
    print(i)

```

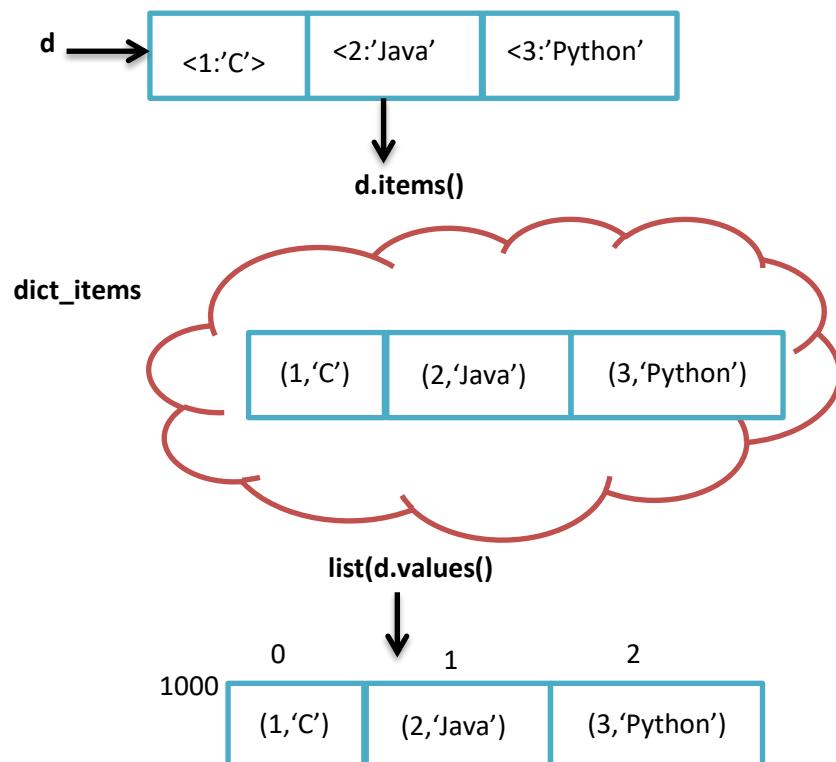
Output:

```

In [11]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
['C', 'Java', 'Python']
C
Java
Python

```

Now our expectation is to access keys as well as values or also called as key value pair



```
d={1:'C',2:'Java',3:'Python'}
print(list(d.items()))
for i,j in d.items():
    print(i,j)
```

Output:

```
In [12]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
[(1, 'C'), (2, 'Java'), (3, 'Python')]
1 C
2 Java
3 Python
```

Let us see what happens if we don't mention anything and try to iterate over **d**

```
d={1:'C',2:'Java',3:'Python'}

for i in d:
    print(i)
```

Output:

```
In [13]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
1
2
3
```

If we don't specify anything, by default it'll iterate over the keys.



Python Fundamentals

day 40

Today's Agenda

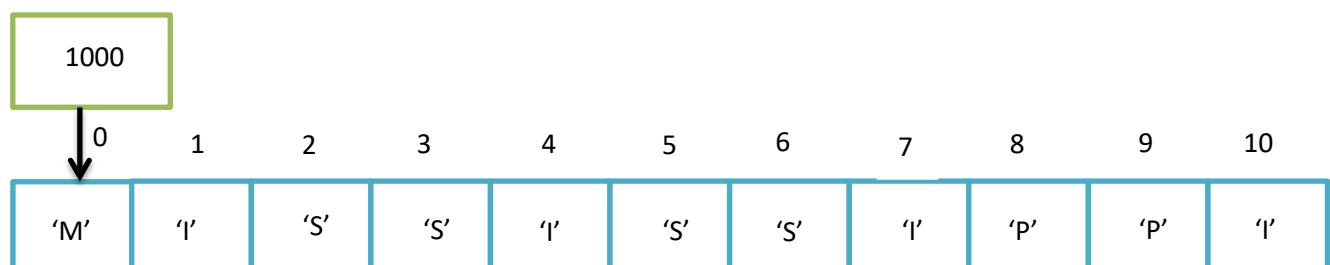
- Programs on dictionary



Programs on dictionary

Example 1: Write a Program to count the occurrence of each character in the given strings.

s



1000

```
s=input("Enter a word\n").upper()
d={}

for i in s:
    if i not in d:
        d[i]=1
    else:
        d[i]=d[i]+1

print(d)
```



Output:

```
In [2]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
```

```
Enter a word  
mississippi  
{'M': 1, 'I': 4, 'S': 4, 'P': 2}
```

To the same example we now want to print those characters which have occurred more than 3 times

```
s=input("Enter a word\n").upper()  
d={}  
  
for i in s:  
    if i not in d:  
        d[i]=1  
    else:  
        d[i]=d[i]+1  
  
for i in d:  
    if d[i]>=3:  
        print(i)
```

d →

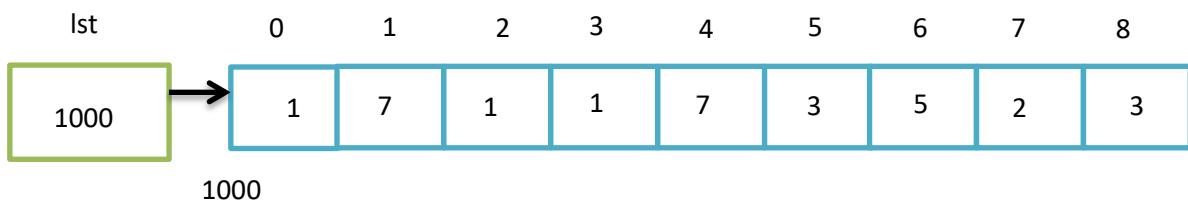
Key	Values
'M'	1
'I'	4
'S'	4
'P'	2

Output:

```
In [3]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
```

```
Enter a word  
mississippi  
I  
S
```

Example 2: Write the program to count total number of pairs.



```

lst=map(int,input().split())
d={}

for i in lst:
    if i not in d:
        d[i]=1
    else:
        d[i] +=1

res=0
for i in d.values():
    res+=i//2
print(res)

```

→

key	value
1	3
7	2
3	2
5	1
2	1

Output:

```
In [4]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
```

```
1 7 1 1 7 3 5 2 3
3
```

Example 3: Write the program to print the mobile number associated with the name, if the name is not among the entry display 'contact not found'.

```

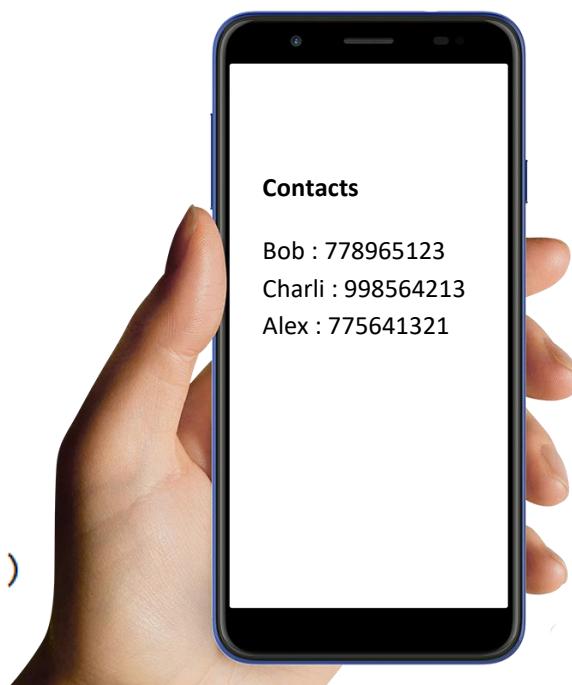
n=int(input())
d={}

for i in range(n):
    l=input().split()
    d[l[0].lower()]=l[1]

s=int(input())

for i in range(s):
    name=input().lower()
    if name in d:
        print('mob:',d[name])
    else:
        print('No contact found')

```



Output:

```
In [7]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')

3
Bob 778965123
Charli 998564213
Alex 775641321

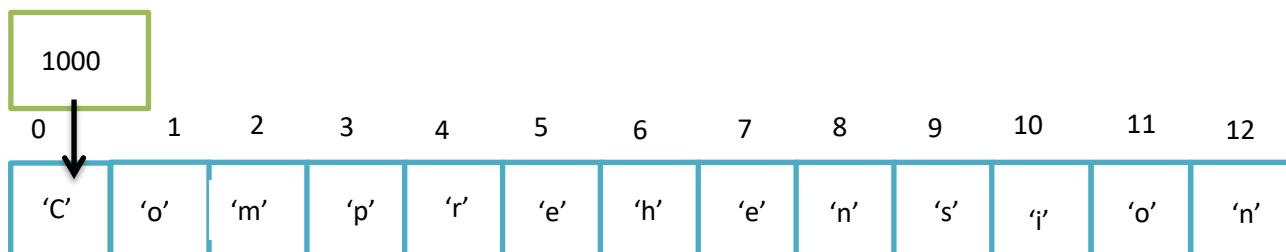
2
bob
mob: 778965123

rohit
No contact found
```



Example 4: Write a program to print the Kth non-repeating character in the given string.

lst



1000

```
lst=input()
k=int(input())
d={}

for i in lst:
    if i not in d:
        d[i]=1
    else:
        d[i]+=1

count=0
for i in d:
    if d[i]==1:
        count+=1
    if k==count:
        print(i)
        break
```



d →

Key	Value
'C'	1
'o'	2
'm'	1
'p'	1
'r'	1
'e'	2
'h'	1
'n'	2
's'	1
'i'	1

Output:

```
In [8]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')

comprehension

3
p
```



Python Fundamentals

day 41

Today's Agenda

- Programs on Dictionary contd...



Programs on Dictionary contd...

Example 1: Write a program to count the occurrence of each word in the given sentence. And print the words that have occurred more than 3 times.

Input sentence: That That Is Is That That Is Not Is Not Is That It It Is

```
import re
s=input().upper()
s=re.sub(r'[.,!?]', '', s)
lst=s.split()
d={}
for i in lst:
    if i in d:
        d[i]+=1
    else:
        d[i]=1
for i in d:
    if d[i]>=3:
        print(i)
```

d →

Key	Value
'THAT'	5
'IS'	6
'NOT'	2
'IT'	2

Output:

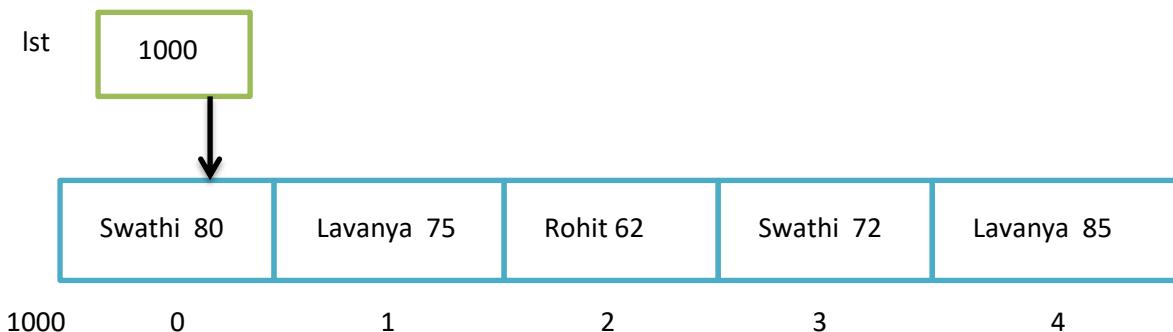
```
In [2]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
```

```
that that is is that that is not is not is that  
it it is  
THAT  
IS
```

Example 2: Write the program to print the highest marks of every person.

Input: Alex 50, Bob 70, Alex 90, Bob 30

Output: Alex 90, Bob 70



```
lst=input().split(',')
d={}

for i in lst:
    t=i.split()
    if t[0] not in d:
        d[t[0]]=t[1]
    else:
        if t[1]>d[t[0]]:
            d[t[0]]=t[1]

print(d)
```



Output:

```
In [4]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
```

```
swathi 80, lavanya 75, rohit 62, swathi 72,  
lavanya 85  
{'swathi': '80', 'lavanya': '85', 'rohit':  
'62'}
```

Example 3: Write a program to inverse the dictionary in such a way keys becomes values and values becomes keys.

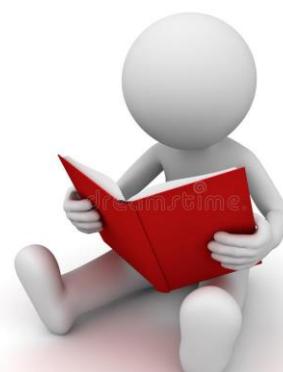
d →

Key	Value
1	'A'
2	'B'
3	'A'
4	'B'
5	'B'
6	'C'

res →

Key	Value
'A'	0 1 1 3
'B'	0 1 2 2 4 5
'C'	0 6

```
d={1:'A',2:'B',3:'A',4:'B',5:'B',6:'C'}  
res=[]  
  
for i in d:  
    if d[i] not in res:  
        res[d[i]]=[]  
        res[d[i]].append(i)  
    else:  
        res[d[i]].append(i)  
  
print(res)
```



Output:

```
In [5]: runfile('C:/Users/rooman/OneDrive/  
Desktop/python/test.py', wdir='C:/Users/rooman/  
OneDrive/Desktop/python')  
{'A': [1, 3], 'B': [2, 4, 5], 'C': [6]}
```



Python Fundamentals

day 42

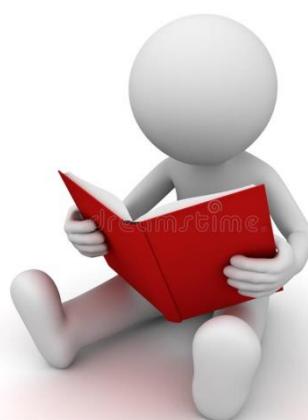
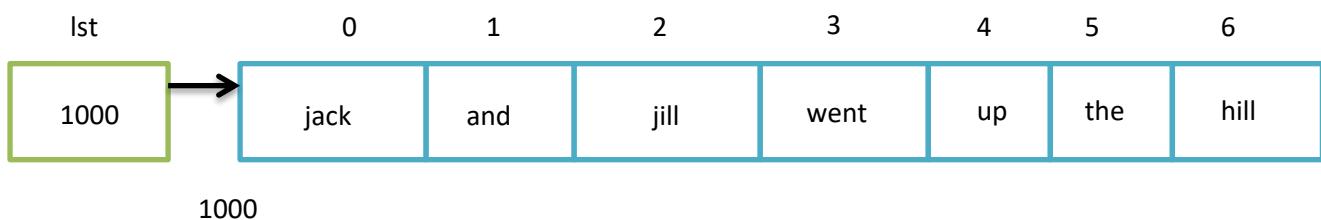
Today's Agenda

- Programs on dictionary contd...
- Dictionary comprehension



Programs on dictionary contd...

Example 1: Write a program that splits the sentence and arrange them into descending order based on their length and then in chronological order.



d →

Key	Value			
	0	1	2	3
4	jack	jill	went	hill
3	and	the		
2	up			

```

lst=input().lower().split()
d={}

for i in lst:
    if len(i) not in d:
        d[len(i)]=[]
        d[len(i)].append(i)
    else:
        d[len(i)].append(i)

s_d=sorted(d.keys(),reverse=True)
for i in s_d:
    for j in sorted(d[i]):
        print(j)

```

Output:

```
In [1]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
```

```

jack and jill went up the hill
hill
jack
jill
went
and
the
up

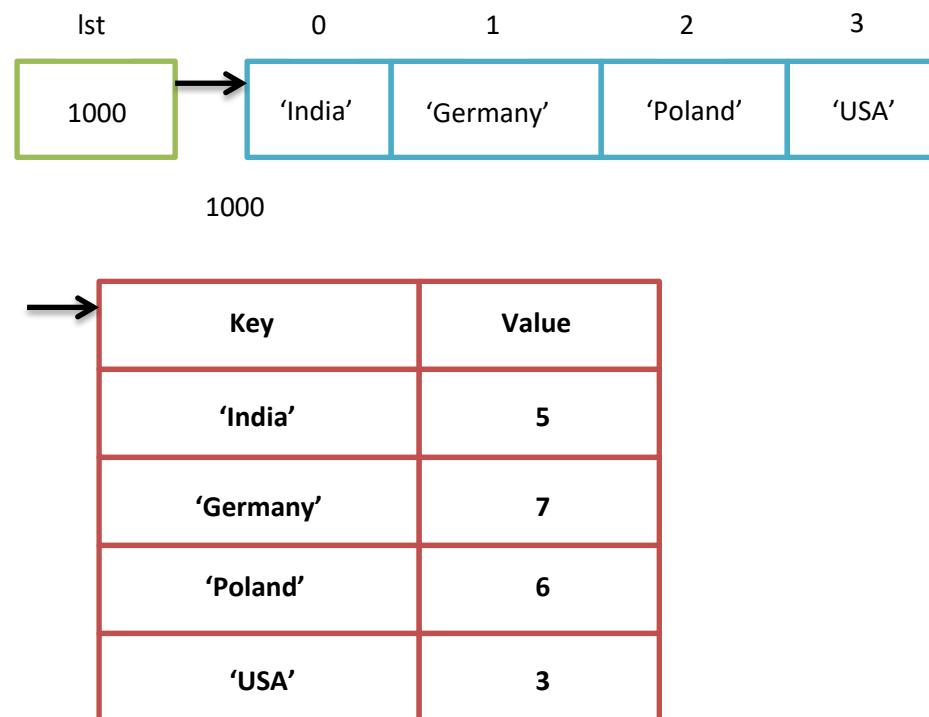
```



Dictionary comprehension

Similar to list comprehension and set comprehension, dictionary comprehension is a concise way of creating dictionaries.

Let us start with an example where we take input from user and want to store it as key and the value should be the length of the word.



```
#traditional way
lst=input().split()
d={}
for i in lst:
    d[i]=len(i)
print(d)
```

Output:

```
In [2]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
```

```
india germany poland usa
{'india': 5, 'germany': 7, 'poland': 6, 'usa': 3}
```

```
#using comprehension
lst=input().split()
d={i:len(i) for i in lst}

print(d)
```

Output:

```
In [3]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')

india germany poland usa
{'india': 5, 'germany': 7, 'poland': 6, 'usa':
3}
```

Example 2: From the given list of numbers take even numbers as the key and square of it as values.

```
#traditional way
lst=[1,2,3,4,5,6,7,8,9]
d={}

for i in lst:
    if i%2==0:
        d[i]=i**2

print(d)
```



Output:

```
In [4]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
{2: 4, 4: 16, 6: 36, 8: 64}
```

```
#using comprehension
lst=[1,2,3,4,5,6,7,8,9]
d={i:i**2 for i in lst if i%2==0}

print(d)
```

Output:

```
In [5]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
{2: 4, 4: 16, 6: 36, 8: 64}
```

Now in the same example along with the even numbers for odd numbers let us print their cube.

```
#traditional way
lst=[1,2,3,4,5,6,7,8,9]
d={}

for i in lst:
    if i%2==0:
        d[i]=i**2
    else:
        d[i]=i**3

print(d)
```



Output:

```
In [6]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
{1: 1, 2: 4, 3: 27, 4: 16, 5: 125, 6: 36, 7: 343, 8: 64, 9: 729}
```

```
#using comprehension
lst=[1,2,3,4,5,6,7,8,9]
d={i : (i**2 if i%2==0 else i**3) for i in lst}

print(d)
```

Output:

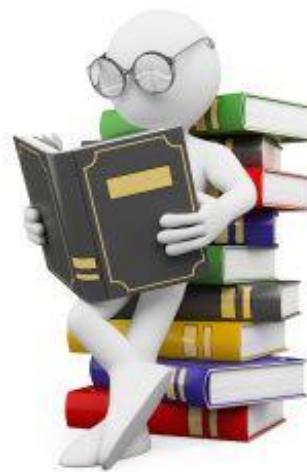
```
In [7]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
{1: 1, 2: 4, 3: 27, 4: 16, 5: 125, 6: 36, 7: 343, 8: 64, 9: 729}
```

Example 3: To the given list check if the length is less than 6 if so convert it into uppercase and cube the value of length and if the length is greater than or equal to 6 then just square the length value.

```
#traditional way
lst=['India','Srilanka','USA','Poland']
d={}

for i in lst:
    if len(i)<6:
        if len(i)%2==0:
            d[i.upper()]=len(i)**2
        else:
            d[i.upper()]=len(i)**3
    else:
        if len(i)%2==0:
            d[i.lower()]=len(i)**2
        else:
            d[i.lower()]=len(i)**3

print(d)
```



Output:

```
In [9]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
{'INDIA': 125, 'srilanka': 64, 'USA': 27,
'poland': 36}
```

```
#using comprehension
lst=['India','Srilanka','USA','Poland']
d={(i.upper() if len(i)<6 else i.lower())
 : (len(i)**2 if len(i)%2==0 else len(i)**3) for i in lst}

print(d)
```

Output:

```
In [10]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
{'INDIA': 125, 'srilanka': 64, 'USA': 27,
'poland': 36}
```

Python Fundamentals

day 43

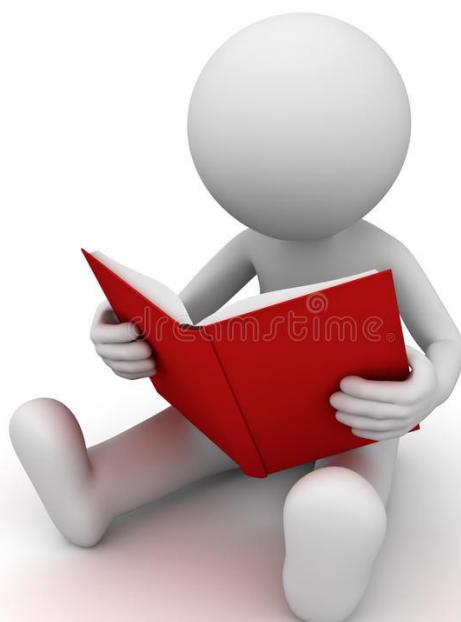
Today's Agenda

- Performance analysis
- Difference between list, set, tuple and dictionary
- Collection module



Performance analysis of dictionary

The performance analysis of dictionary is similar to that of sets. As both of them use hashing function to store the data. Kindly refer back if you don't remember.



Difference between list, set, tuple and dictionary

List	Set	Dictionary	Tuple
<code>lst=[10,12,15]</code>	<code>st={1,23,24}</code>	<code>dict={1:32,2:45}</code>	<code>tup=("spam",55)</code>
<code>print(lst[0])</code>	<code>print(st)</code> Set elements can't be indexed.	<code>print(dict[1])</code>	<code>print(tup[0])</code>
Can contain duplicate elements.	Can't contain duplicate elements. Faster compared to list	Can't contain duplicate keys, but can contain duplicate values	Can contain duplicate elements. Faster compared to list
Mutable	Mutable	Mutable	Immutable, Values can't be changed ones assigned.
Slicing can be done	Slicing can't be done(unordered data)	Slicing can't be done	Slicing can be done
Memory inefficient	Memory inefficient	Memory inefficient	Memory efficient
Slow creation	Slow creation	Slow creation	Fast creation
Usage: Use list if you have a collection of data that doesn't need random access. Use lists when you need a simple, iterable collection that is modified frequently.	Usage: Membership testing and the elimination of duplicate entries. When you need uniqueness for the elements.	Usage: When you need logical association between key:value pair. When you need fast lookup for your data, based on a custom key. When your data is being constantly modified.	Usage: Use tuples when your data cannot change. A tuple is used in combination with a dictionary. For example a tuple might represent a key because it's immutable.

Collection module

Collection module in python has certain classes like **chainmap**, **counter**, **namedtuple** and **deque** these classes are very effective and great tools to work with. Let us start with **chainmap**

To understand the use of chainmap, let us consider the following example of hypermarket where you find all the necessary items for day to day life.

```
clothes={'shirts':100,'pants':85,'blazers':35}
ele={'mobiles':150,'computers':40,'AC':60}
food={'milk':200,'chips':180,'biscuit':280}

inv={**clothes, **ele, **food} #merging of dictionaries
print(inv)
print(inv['computers'])
inv['shirts']=95
print(inv)
print(clothes)
```

Output:

```
In [1]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
{'shirts': 100, 'pants': 85, 'blazers': 35,
'mobiles': 150, 'computers': 40, 'AC': 60,
'milk': 200, 'chips': 180, 'biscuit': 280}
40
{'shirts': 95, 'pants': 85, 'blazers': 35,
'mobiles': 150, 'computers': 40, 'AC': 60,
'milk': 200, 'chips': 180, 'biscuit': 280}
{'shirts': 100, 'pants': 85, 'blazers': 35}
```

We see that the value of shirts is not reflected in the original dictionary. To make this happen we should use the class chainmap present in collection module as below

```
from collections import ChainMap

clothes={'shirts':100,'pants':85,'blazers':35}
ele={'mobiles':150,'computers':40,'AC':60}
food={'milk':200,'chips':180,'biscuit':280}

inv=ChainMap(clothes, ele, food) #overview of dictionaries

inv['shirts']=95
print(inv)
print(clothes)
```

Output:

```
In [2]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
ChainMap({'shirts': 95, 'pants': 85, 'blazers': 35}, {'mobiles': 150, 'computers': 40, 'AC': 60}, {'milk': 200, 'chips': 180, 'biscuit': 280})
{'shirts': 95, 'pants': 85, 'blazers': 35}
```

Now we can see the changes being reflected both in both the dictionaries.

Let us now see what happens if we try to add an element

```
from collections import ChainMap

clothes={'shirts':100,'pants':85,'blazers':35}
ele={'mobiles':150,'computers':40,'AC':60}
food={'milk':200,'chips':180,'biscuit':280}

inv=ChainMap(clothes, ele, food) #overview of dictionaries

inv['jeans']=55#adding new element
print(inv)
print(clothes)
```

Output:

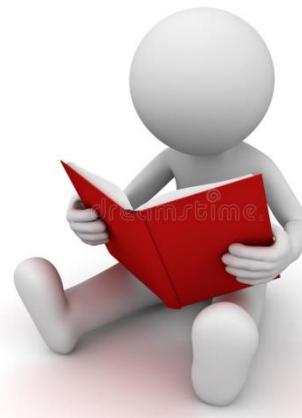
```
In [3]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
ChainMap({'shirts': 100, 'pants': 85, 'blazers': 35, 'jeans': 55}, {'mobiles': 150, 'computers': 40, 'AC': 60}, {'milk': 200, 'chips': 180, 'biscuit': 280})
{'shirts': 100, 'pants': 85, 'blazers': 35, 'jeans': 55}
```

New element always gets added in the first dictionary.

Let us now look at the next class that is **counter**

```
s="mississippi"
d={}
for i in s:
    if i not in d:
        d[i]=1
    else:
        d[i]+=1

print(d)
```



Output:

```
In [4]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
{'m': 1, 'i': 4, 's': 4, 'p': 2}
```

The above code can be written in a single line using counter class.

```
from collections import Counter

s="mississippi"
c=Counter(s)
print(c)
```

Output:

```
In [5]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
Counter({'i': 4, 's': 4, 'p': 2, 'm': 1})
```

Whenever you want to count the number of occurrences and store it as a dictionary then always make use of counter class.

Next let us see the different ways of using counter class

```
from collections import Counter

#passing list
c=Counter(['a','a','b','c','b','c','b','b','a'])
print(c)
#passing set
c=Counter({'a','a','b','c','b','c','b','b','a'})
print(c)
#passing keyword arguments
c=Counter(a=2,b=4,c=2)
print(c)
#passing dictionary
c=Counter({'a':2,'b':4,'c':2})
print(c)
```

Output:

```
In [6]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
Counter({'b': 4, 'a': 3, 'c': 2})
Counter({'b': 1, 'a': 1, 'c': 1})
Counter({'b': 4, 'a': 2, 'c': 2})
Counter({'b': 4, 'a': 2, 'c': 2})
```

Below are some of the methods commonly used in counter class

```
from collections import Counter

c=Counter(['a','a','b','c','b','c','b','b','a'])
print(list(c.elements())) #show all the elements
#gives the element that has occurred the most along with count
print(c.most_common(1))
```

Output:

```
In [7]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
['a', 'a', 'a', 'b', 'b', 'b', 'b', 'c', 'c']
[('b', 4)]
```

Next let us see some operation which can be performed on counter objects

```
from collections import Counter

c1=Counter(a=4,b=3,c=2)
c2=Counter(a=3,b=2,c=1)

#addition
c3=c1+c2
print(c3)
#subtraction
c3=c1-c2
print(c3)
#Union- gives the maximum value
c3=c1|c2
print(c3)
#Intersection - gives the minimum value
c3=c1&c2
print(c3)
```



Output:

```
In [8]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
Counter({'a': 7, 'b': 5, 'c': 3})
Counter({'a': 1, 'b': 1, 'c': 1})
Counter({'a': 4, 'b': 3, 'c': 2})
Counter({'a': 3, 'b': 2, 'c': 1})
```

Moving ahead let us look into **namedTuple**

Named tuples assign meaning to each position in a tuple and allow for more readable, self-documenting code. They can be used wherever regular tuples are used, and they add the ability to access fields by name instead of position index. Let us look at the example

```
from collections import namedtuple

S=namedtuple('Student',('name','age','stream','avg'))
s1=S('Alex',22,"CS",78)
print(s1)
print(s1[0])
print(s1.name)
#s1[0]='Bob' ---- Will throw error
```

Output:

```
In [9]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
Student(name='Alex', age=22, stream='CS',
avg=78)
Alex
Alex
```

Last but not the least let us look at deque class which stands for double ended queue

```
lst=[10,20,30,40]
lst.append(50)
lst.insert(0,99)
print(lst)
```

Output:

```
In [11]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
[99, 10, 20, 30, 40, 50]
```

The above operation will take linear time. Whereas using deque it can be done in constant time

```
from collections import deque

dq=deque([10,20,30,40])
dq.append(50)
dq.appendleft(99)
print(dq)
```

Output:

```
In [10]: runfile('C:/Users/rooman/OneDrive/
Desktop/python/test.py', wdir='C:/Users/rooman/
OneDrive/Desktop/python')
deque([99, 10, 20, 30, 40, 50])
```



Python Fundamentals

day 44

Today's Agenda

- Object orientation



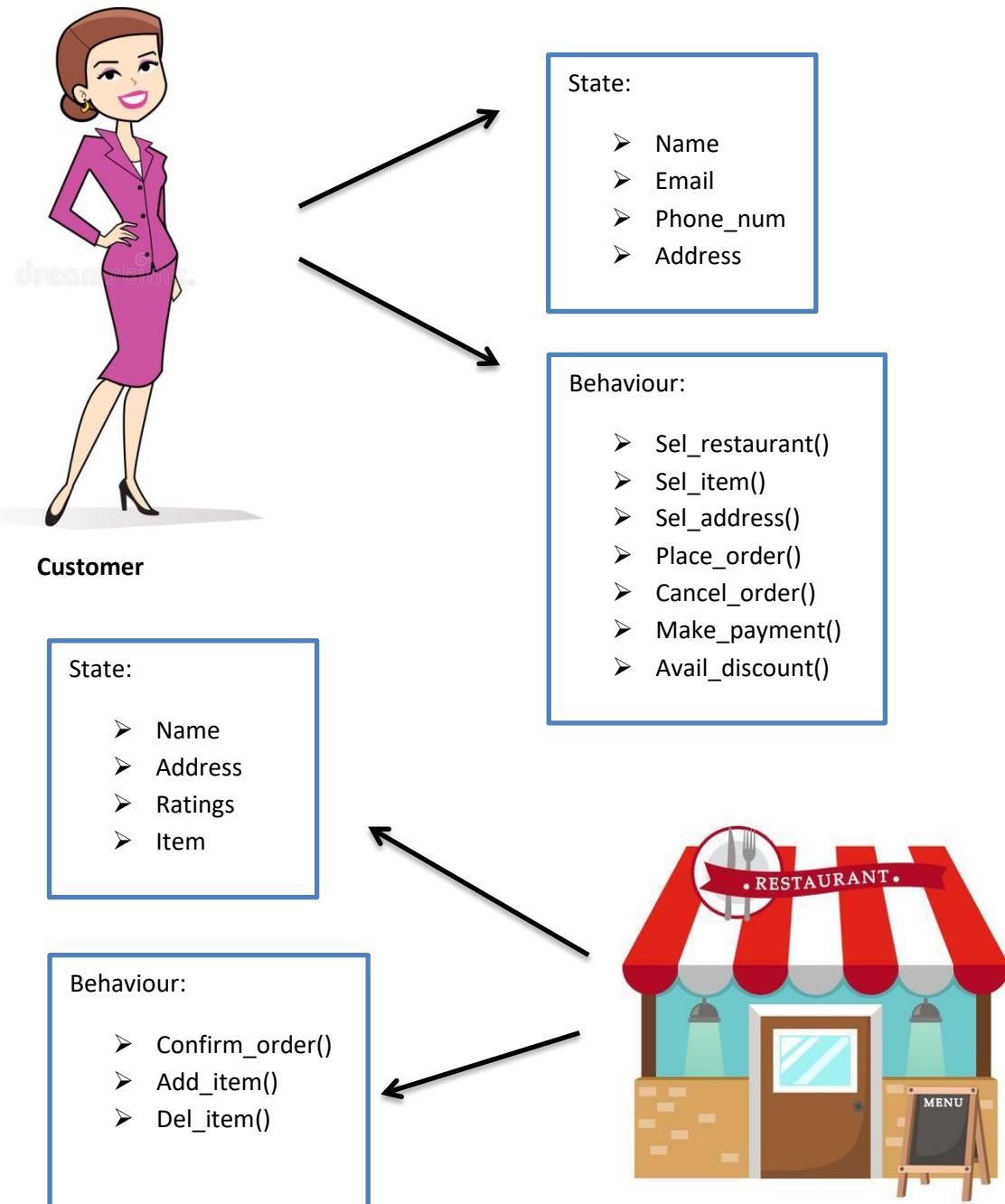
Object orientation

So far we have seen the structured style of programming language where we used different functions to perform operations. Now it's time to have a look at providing solutions using the classes and the objects.

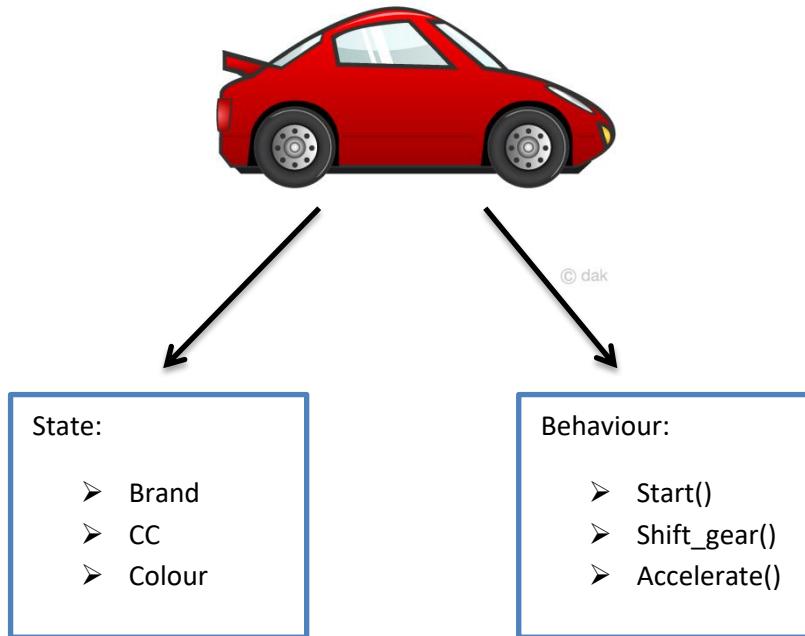
For example: Imagine you are standing in a queue for getting the food every time. So suddenly you have a eureka moment and plan to have a mobile application into which all the food items are present. All you have to do is click on whichever food item you like and get it delivered at your doorstep. So this person goes to a software company and proposes this idea, where upon clicking on the food item it should ask from which store you want to purchase it and then track down your location after which mode of payment is selected and within few minutes you can enjoy the food at your desired location.

For all this to work software developer should see this in an object oriented perspective where different objects are interacting with each other, in this case customer, restaurant and delivery agent are the objects interacting with one another via app.

For software to recognize real time entities as objects it should know the state and behaviour of objects.



Now let us see how to implement these in python considering the following example



```
class Car():
    #states of object
    def __init__(self):
        self.brand='BMW'
        self.cc=2100
        self.color='blue'

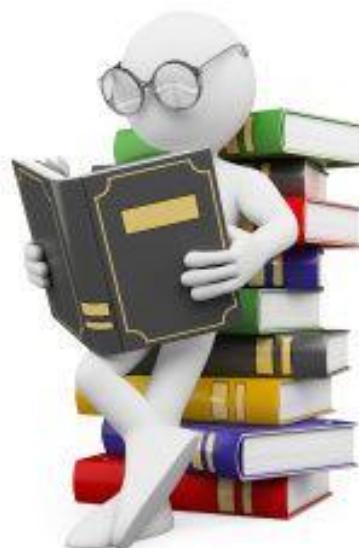
    #behaviours of object
    def start_engine(self):
        print('engine is starting')

    def shift_gear(self):
        print('shifting gear')

    def accelerate(self):
        print('car is accelerating')

    def main():
        c1=Car()
        print(c1.brand)
        print(c1.cc)
        print(c1.color)
        c1.start_engine()
        c1.shift_gear()
        c1.accelerate()

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



Output:

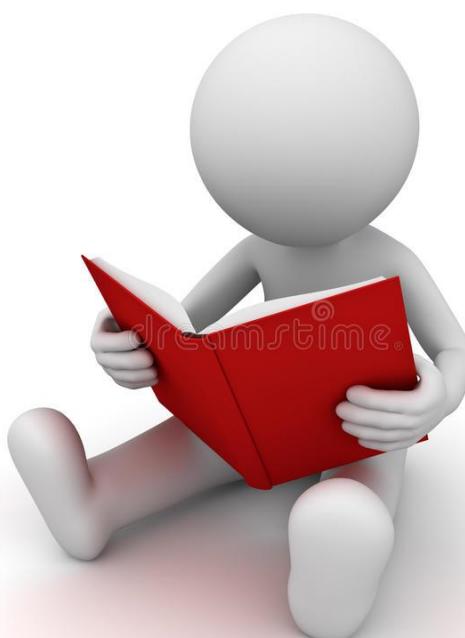
```
In [3]: runfile('C:/Users/rooman/OneDrive/  
Desktop/python/test.py', wdir='C:/Users/rooman/  
OneDrive/Desktop/python')  
BMW  
2100  
blue  
engine is starting  
shifting gear  
car is accelerating
```

You must be wondering why should we pass self as the argument and is it mandatory to use it?



Certainly using self as argument is mandatory because internally c1 object is passed as argument which contains the different states of object. So, not using self will throw an error. But replacing self with any other character will definitely work. The only issue will be not following the conventions of python.

We know that there exist multiple objects in real life. So let us see if the above code could have another object in it



```

class Car():
    #states of object
    def __init__(self):
        self.brand='BMW'
        self.cc=2100
        self.color='blue'

    #behaviours of object
    def start_engine(self):
        print('engine is starting')

    def shift_gear(self):
        print('shifting gear')

    def accelerate(self):
        print('car is accelerating')

def main():
    c1=Car()
    print(c1.brand)
    print(c1.cc)
    print(c1.color)
    c1.start_engine()
    c1.shift_gear()
    c1.accelerate()

    c2=Car()
    print(c2.brand)
    print(c2.cc)
    print(c2.color)
    c2.start_engine()
    c2.accelerate()
    c2.shift_gear()

if __name__=='__main__':
    main()

```



Object:

```

In [4]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
BMW
2100
blue
engine is starting
shifting gear
car is accelerating
BMW
2100
blue
engine is starting
car is accelerating
shifting gear

```

Python Fundamentals

day 45

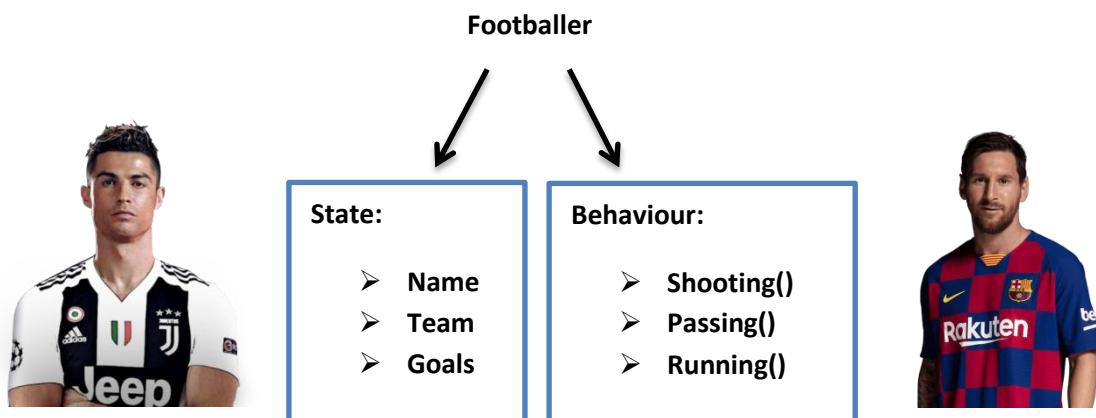
Today's Agenda

- Object orientation contd...
- Creating instance variables



Object orientation contd...

In previous session we got to know multiple objects can be created. But we had created duplicate objects. So let us now consider an example where we shall have different objects of the same type.



Let us see how to write the python code for this particular example.

```

class FootBaller:
    def __init__(self, name, team, goals):
        self.name=name
        self.team=team
        self.goals=goals

    def shooting(self):
        print(self.name,'is shooting')

    def passing(self):
        print(self.name,'is passing')

    def running(self):
        print(self.name,'is running')

def main():
    cr=FootBaller('Cristino','Juventus','746')
    print(cr.name)
    print(cr.team)
    print(cr.goals)
    cr.shooting()
    cr.passing()
    cr.running()

    messi=FootBaller('Messi','Barcelona','700')
    print(messi.name)
    print(messi.team)
    print(messi.goals)
    messi.shooting()
    messi.passing()
    messi.running()

if __name__ == '__main__':
    main()

```

Output:

```

In [2]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
Cristino
Juventus
746
Cristino is shooting
Cristino is passing
Cristino is running
Messi
Barcelona
700
Messi is shooting
Messi is passing
Messi is running

```



Great!! But we Python is all above reducing lines of code and making it more efficient. So let us see how more efficient form of the above code.

```
class FootBaller:  
    def __init__(self, name, team, goals):  
        self.name=name  
        self.team=team  
        self.goals=goals  
  
    def shooting(self):  
        print(self.name, 'is shooting')  
  
    def passing(self):  
        print(self.name, 'is passing')  
  
    def running(self):  
        print(self.name, 'is running')  
  
    def display(self):  
        print(self.name)  
        print(self.team)  
        print(self.goals)  
  
def main():  
    cr=FootBaller('Cristino', 'Juventus', '746')  
    cr.display()  
    cr.shooting()  
    cr.passing()  
    cr.running()  
  
    messi=FootBaller('Messi', 'Barcelona', '700')  
    messi.display()  
    messi.shooting()  
    messi.passing()  
    messi.running()  
  
if __name__ == '__main__':  
    main()
```

Output:

```
In [3]: runfile('C:/Users/rooman/OneDrive/Desktop/python/  
test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')  
Cristino  
Juventus  
746  
Cristino is shooting  
Cristino is passing  
Cristino is running  
Messi  
Barcelona  
700  
Messi is shooting  
Messi is passing  
Messi is running
```



From the above example we got to know the following things

- ❖ For the given class we can create multiple objects. And objects can be initialised differently.
- ❖ There are two different types of functions
 1. `__new__` : Is a constructor.
 2. `__init__` : Is a initialiser.

Creating instance variables

There are two ways in which we can create instance variables.

- ❖ During object creation
- ❖ After object creation

```
class FootBaller:  
    def __init__(self, name, team, goals): # before object creation  
        self.name=name  
        self.team=team  
        self.goals=goals  
  
    def shooting(self):  
        print(self.name, 'is shooting')  
  
    def passing(self):  
        print(self.name, 'is passing')  
  
    def running(self):  
        print(self.name, 'is running')  
  
    def display(self):  
        print(self.name)  
        print(self.team)  
        print(self.goals)  
        print(self.age)  
        print(self.jersey_no)  
  
    def main():  
        cr=FootBaller('Cristino', 'Juventus', '746')  
        cr.age=35 #after object creation  
        cr.jersey_no=7 #after object creation  
        cr.display()  
        cr.shooting()  
        cr.passing()  
        cr.running()  
  
if __name__ == '__main__':  
    main()
```

Output:

```
In [4]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
Cristino
Juventus
746
35
7
Cristino is shooting
Cristino is passing
Cristino is running
```

There is also another way of creating instance variables after object creation, by using built-in method **setattr()**. Similarly we also have **getattr()** and **hasattr()**. Let us see how to use it

```
class FootBaller:
    def __init__(self, name, team, goals): # before object creation
        self.name=name
        self.team=team
        self.goals=goals

    def shooting(self):
        print(self.name, 'is shooting')

    def passing(self):
        print(self.name, 'is passing')

    def running(self):
        print(self.name, 'is running')

    def display(self):
        print(self.name)
        print(self.team)
        print(self.goals)
        print(self.age)
        print(self.jersey_no)

def main():
    cr=FootBaller('Cristino', 'Juventus', '746')
    setattr(cr, 'age', 35) #after object creation
    setattr(cr, 'jersey_no', 7)#after object creation
    print(cr.name)
    print(getattr(cr, 'name'))
    print(hasattr(cr, 'name'))
    print(hasattr(cr, 'gender'))

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



Output:

```
In [6]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
Cristino
Cristino
True
False
```

Another interesting thing to know is all the instance variables are present in a dictionary. The name of this dictionary is __dict__. Let us see if this is actually true.

```
class FootBaller:
    def __init__(self, name, team, goals): # before
        self.name=name
        self.team=team
        self.goals=goals

    def shooting(self):
        print(self.name, 'is shooting')

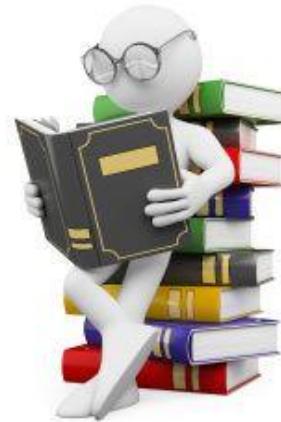
    def passing(self):
        print(self.name, 'is passing')

    def running(self):
        print(self.name, 'is running')

    def display(self):
        print(self.name)
        print(self.team)
        print(self.goals)
        print(self.age)
        print(self.jersey_no)

def main():
    cr=FootBaller('Cristino', 'Juventus', '746')
    setattr(cr, 'age', 35) # after object creation
    setattr(cr, 'jersey_no', 7) # after object creation
    print(cr.__dict__) # printing the dictionary
    print(cr.name)
    print(cr.__dict__['name'])

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



Output:

```
In [7]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
{'name': 'Cristino', 'team': 'Juventus', 'goals': '746',
'age': 35, 'jersey_no': 7}
Cristino
Cristino
```



Python Fundamentals

day 45

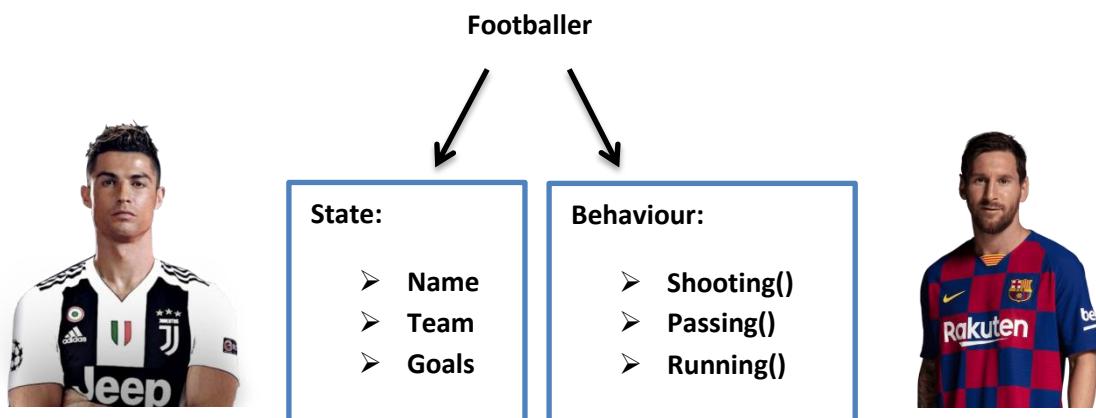
Today's Agenda

- Object orientation contd...
- Creating instance variables



Object orientation contd...

In previous session we got to know multiple objects can be created. But we had created duplicate objects. So let us now consider an example where we shall have different objects of the same type.



Let us see how to write the python code for this particular example.

```

class FootBaller:
    def __init__(self, name, team, goals):
        self.name=name
        self.team=team
        self.goals=goals

    def shooting(self):
        print(self.name,'is shooting')

    def passing(self):
        print(self.name,'is passing')

    def running(self):
        print(self.name,'is running')

def main():
    cr=FootBaller('Cristino','Juventus','746')
    print(cr.name)
    print(cr.team)
    print(cr.goals)
    cr.shooting()
    cr.passing()
    cr.running()

    messi=FootBaller('Messi','Barcelona','700')
    print(messi.name)
    print(messi.team)
    print(messi.goals)
    messi.shooting()
    messi.passing()
    messi.running()

if __name__ == '__main__':
    main()

```

Output:

```

In [2]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
Cristino
Juventus
746
Cristino is shooting
Cristino is passing
Cristino is running
Messi
Barcelona
700
Messi is shooting
Messi is passing
Messi is running

```



Great!! But we Python is all above reducing lines of code and making it more efficient. So let us see how more efficient form of the above code.

```
class FootBaller:  
    def __init__(self, name, team, goals):  
        self.name=name  
        self.team=team  
        self.goals=goals  
  
    def shooting(self):  
        print(self.name, 'is shooting')  
  
    def passing(self):  
        print(self.name, 'is passing')  
  
    def running(self):  
        print(self.name, 'is running')  
  
    def display(self):  
        print(self.name)  
        print(self.team)  
        print(self.goals)  
  
def main():  
    cr=FootBaller('Cristino', 'Juventus', '746')  
    cr.display()  
    cr.shooting()  
    cr.passing()  
    cr.running()  
  
    messi=FootBaller('Messi', 'Barcelona', '700')  
    messi.display()  
    messi.shooting()  
    messi.passing()  
    messi.running()  
  
if __name__ == '__main__':  
    main()
```

Output:

```
In [3]: runfile('C:/Users/rooman/OneDrive/Desktop/python/  
test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')  
Cristino  
Juventus  
746  
Cristino is shooting  
Cristino is passing  
Cristino is running  
Messi  
Barcelona  
700  
Messi is shooting  
Messi is passing  
Messi is running
```



From the above example we got to know the following things

- ❖ For the given class we can create multiple objects. And objects can be initialised differently.
- ❖ There are two different types of functions
 1. `__new__` : Is a constructor.
 2. `__init__` : Is a initialiser.

Creating instance variables

There are two ways in which we can create instance variables.

- ❖ During object creation
- ❖ After object creation

```
class FootBaller:  
    def __init__(self, name, team, goals): # before object creation  
        self.name=name  
        self.team=team  
        self.goals=goals  
  
    def shooting(self):  
        print(self.name, 'is shooting')  
  
    def passing(self):  
        print(self.name, 'is passing')  
  
    def running(self):  
        print(self.name, 'is running')  
  
    def display(self):  
        print(self.name)  
        print(self.team)  
        print(self.goals)  
        print(self.age)  
        print(self.jersey_no)  
  
    def main():  
        cr=FootBaller('Cristino', 'Juventus', '746')  
        cr.age=35 #after object creation  
        cr.jersey_no=7 #after object creation  
        cr.display()  
        cr.shooting()  
        cr.passing()  
        cr.running()  
  
if __name__ == '__main__':  
    main()
```

Output:

```
In [4]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
Cristino
Juventus
746
35
7
Cristino is shooting
Cristino is passing
Cristino is running
```

There is also another way of creating instance variables after object creation, by using built-in method **setattr()**. Similarly we also have **getattr()** and **hasattr()**. Let us see how to use it

```
class FootBaller:
    def __init__(self, name, team, goals): # before object creation
        self.name=name
        self.team=team
        self.goals=goals

    def shooting(self):
        print(self.name, 'is shooting')

    def passing(self):
        print(self.name, 'is passing')

    def running(self):
        print(self.name, 'is running')

    def display(self):
        print(self.name)
        print(self.team)
        print(self.goals)
        print(self.age)
        print(self.jersey_no)

def main():
    cr=FootBaller('Cristino', 'Juventus', '746')
    setattr(cr, 'age', 35) #after object creation
    setattr(cr, 'jersey_no', 7)#after object creation
    print(cr.name)
    print(getattr(cr, 'name'))
    print(hasattr(cr, 'name'))
    print(hasattr(cr, 'gender'))

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



Output:

```
In [6]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
Cristino
Cristino
True
False
```

Another interesting thing to know is all the instance variables are present in a dictionary. The name of this dictionary is __dict__. Let us see if this is actually true.

```
class FootBaller:
    def __init__(self, name, team, goals): # before
        self.name=name
        self.team=team
        self.goals=goals

    def shooting(self):
        print(self.name, 'is shooting')

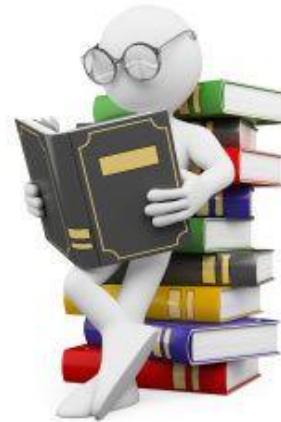
    def passing(self):
        print(self.name, 'is passing')

    def running(self):
        print(self.name, 'is running')

    def display(self):
        print(self.name)
        print(self.team)
        print(self.goals)
        print(self.age)
        print(self.jersey_no)

def main():
    cr=FootBaller('Cristino', 'Juventus', '746')
    setattr(cr, 'age', 35) # after object creation
    setattr(cr, 'jersey_no', 7) # after object creation
    print(cr.__dict__) # printing the dictionary
    print(cr.name)
    print(cr.__dict__['name'])

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



Output:

```
In [7]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
{'name': 'Cristino', 'team': 'Juventus', 'goals': '746',
'age': 35, 'jersey_no': 7}
Cristino
Cristino
```



Python Fundamentals

day 47

Today's Agenda

- Functions as objects
- Decorators



Functions as objects

One of the most powerful features of Python is that **everything is an object, including functions**. Functions in Python are **first-class objects**. Everything a regular object is capable of doing such as integer, floating point numbers, complex numbers, list, set, tuple, dictionaries etc can also be done using functions.

First-class
objects??

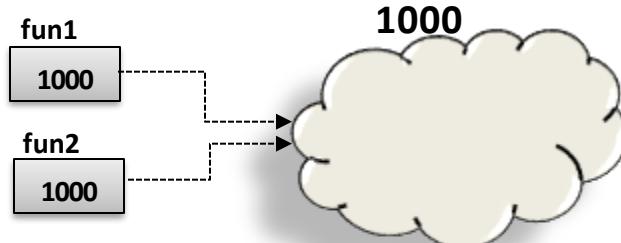
Let us now understand first-class objects with the help of code shown below.



CODE:

```
def fun1():
    print('Inside fun1( )')

fun1()
print(fun1)
fun2 = fun1
fun2()
print(fun2)
```



OUTPUT:

```
Inside fun1( )
<function fun1 at 0x000002008C3AC6A8>
Inside fun1( )
<function fun1 at 0x000002008C3AC6A8>
```

We see in the above output, when we simply print fun1 and fun2, we get the **address of function object as output** which means fun1 and fun2 are the reference variables pointing to the same object.

Can one function be sent as argument to another function?

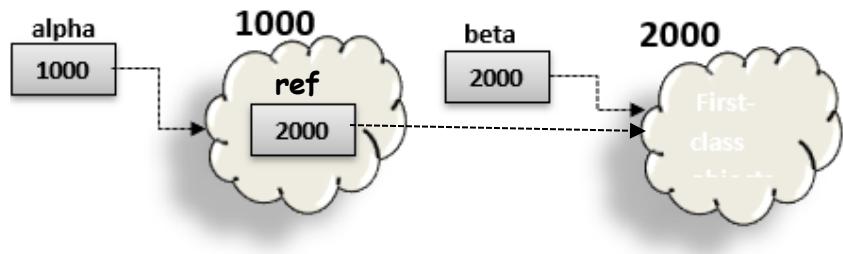
Let us now see can functions be passed as arguments in python?

CODE:

```
def alpha(ref):
    print('Inside alpha( )')
    ref()

def beta():
    print('Inside beta( )')

alpha(beta)
```



OUTPUT:

```
Inside alpha( )
Inside beta( )
```

As we can see from the above output, beta () function is passed as input to alpha () function which concludes **one function can be passed as input to other function in python.**

Can a function be passed as output from another function?

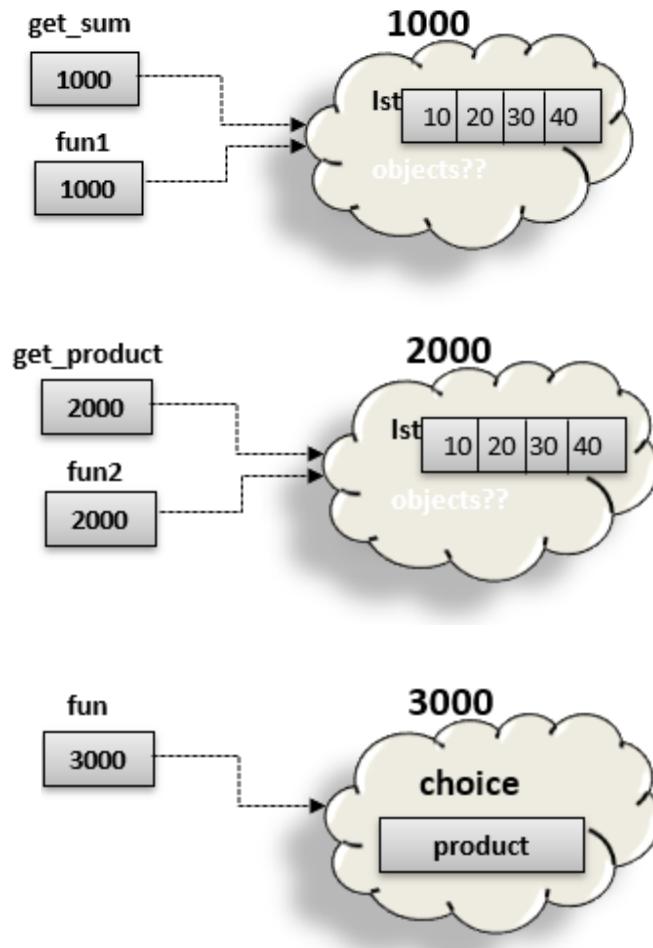
Let us now see can functions be passed as output in python?

CODE:

```
def get_sum(lst):
    print(sum(lst))

def get_product(lst):
    p = 1
    for i in lst:
        p *= i
    print(p)

def fun(choice):
    if choice == 'sum':
        return get_sum
    else:
        return get_product
fun1 = fun('sum')
fun1([10,20,30,40])
fun2 = fun('product')
fun2([10,20,30,40])
```



OUTPUT:

100
240000



From the above output, we can conclude that not only can functions be passed as input but also can be returned as output from another function.

Can one function be present within another function?

A function can be present within another function in python and such functions are only called as **Inner functions**.

An **inner function** is simply a function that is defined **inside** another function. The **inner function** is able to access the variables that have been defined within the scope of the **outer function**, but it cannot change them.

We can call an inner function in two ways as shown below:

CODE:

```
def outer():
    print('Inside outer()')

    def inner():
        print('Inside inner()')
    inner()

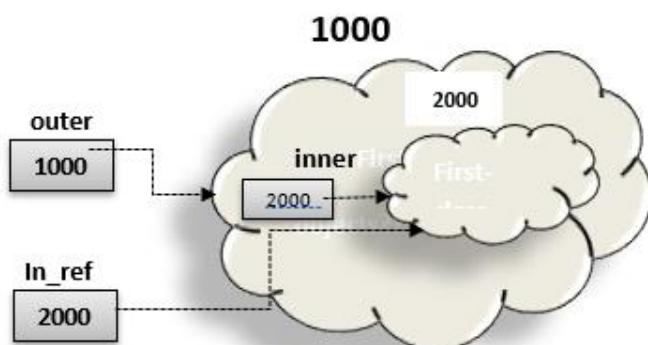
outer()
```

Or

```
def outer():
    print('Inside outer()')

    def inner():
        print('Inside inner()')
    return inner
```

```
in_ref = outer()
in_ref()
```



OUTPUT:

Inside outer()
Inside inner()

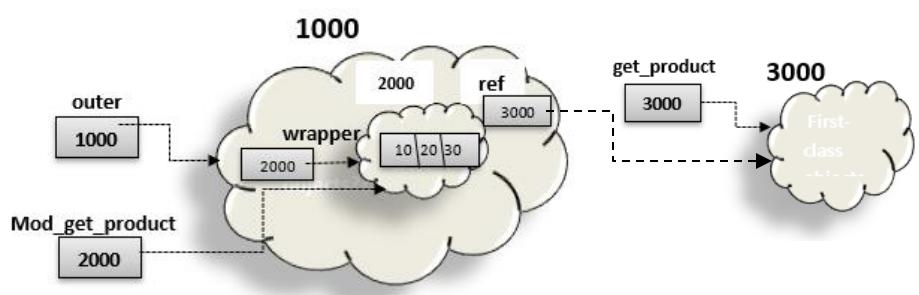
After getting to know all the important points about functions as objects in python, let us now try to code a scenario where we have a `get_product()` which calculates the product of all numbers present in a list but condition is product should not be calculated if 0 is present in list without modifying `get_product()`.

Now how do we achieve it?? Have a look at the code given below!

CODE:

```
def outer(ref):  
  
    def wrapper(lst):  
        if 0 in lst:  
            print('0 is present')  
        else:  
            ref(lst)  
        return wrapper  
  
    def get_product(lst):  
        p = 1  
        for i in lst:  
            p *= i  
        print(p)
```

```
mod_get_product = outer(get_product)  
mod_get_product([10,20,30])  
mod_get_product([10,0,30])
```



OUTPUT:

6000
0 is present

The above output can also be achieved by using the concept of **decorators** in python.

Decorators in python

Decorators in Python allows programmers to modify the behaviour of function or class. Decorators allow us to wrap another function in order to extend the behaviour of wrapped function, **without permanently modifying it**. In Decorators, functions are taken as the argument into another function and then called inside the wrapper function.



Let us now understand how can we achieve the above output using decorators.

CODE:

```
def outer(ref):
```

```
    def wrapper(lst):
        if 0 in lst:
            print('0 is present')
        else:
            ref(lst)
    return wrapper
```

```
@outer
```

```
def get_product(lst):
```

```
    p = 1
    for i in lst:
        p *= i
    print(p)
```



```
get_product([10,20,30])
get_product([10,0,30])
```

OUTPUT:

6000
0 is present



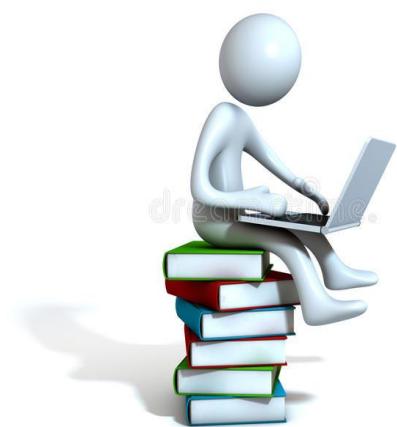
Let's jump to another example to understand decorators in detail.

CODE:

```
def outer(ref):
    def wrapper(a,b):
        if b == 0:
            print("Please provide a non zero denominator")
        else:
            ref(a,b)
    return wrapper

@outer
def div(a,b):
    print(a/b)

#mod_div = outer(div)
#mod_div(10,2)
#mod_div(10,0)
div(10,2)
div(10,0)
```



OUTPUT:

5.0

Please provide a non zero denominator



DID YOU KNOW??

Python is one of the official programming languages at Google and YouTube is one of Google's products that are powered by Python.



Python fundamentals

day 48

Today's Agenda

- Decorators contd...
- Closures



Decorators contd...

Now our expectation is to take an input list and first find the square of each element and then find the product of it. Let's see how to do it

```
def power_of(ref):  
  
    def wrapper(lst):  
        lst=list(map(lambda x:x**2, lst))  
        ref(lst)  
    return wrapper  
  
def get_product(lst):  
    p=1  
    for i in lst:  
        p*=i  
    print(p)  
  
mod_get_product=power_of(get_product)  
mod_get_product([1,2,3,4,5])
```



Output:

```
In [2]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
14400
```

We can do the same in a simpler way

```
def power_of(ref):

    def wrapper(lst):
        lst=list(map(lambda x:x**2, lst))
        ref(lst)
    return wrapper

@power_of
def get_product(lst):
    p=1
    for i in lst:
        p*=i
    print(p)

get_product([1,2,3,4,5])
```

Output:

```
In [4]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
14400
```

`power_of` will automatically consider the power to be 2. But what if we want it to be any other number? Let us see



```

def decorator(num):

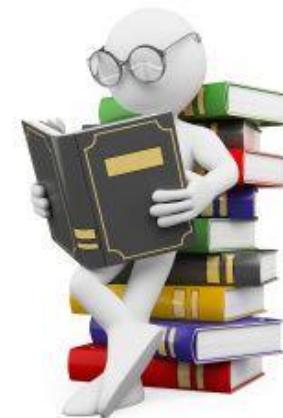
    def power_of(ref):

        def wrapper(lst):
            lst=list(map(lambda x:x**num, lst))
            ref(lst)
        return wrapper
    return power_of

def get_product(lst):
    p=1
    for i in lst:
        p*=i
    print(p)

fun1=decorator(3)
mod_get_product=fun1(get_product)
mod_get_product([1,2,3,4,5])

```

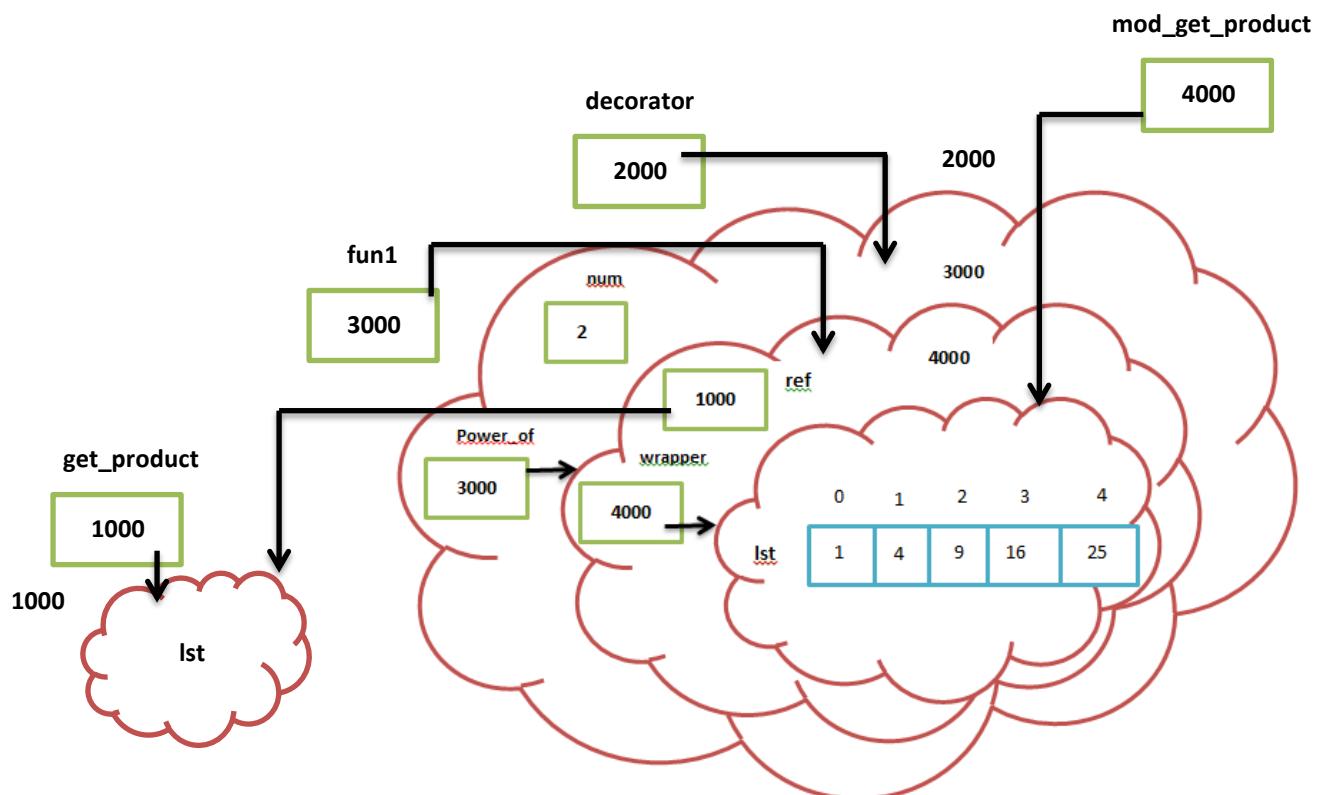


Output:

```

In [10]: runfile('C:/Users/rooman/OneDrive/Desktop/python/
test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
1728000

```



Above code can also be written as

```
def decorator(num):  
    def power_of(ref):  
        def wrapper(lst):  
            lst=list(map(lambda x:x**num, lst))  
            ref(lst)  
        return wrapper  
    return power_of  
  
@decorator(3)  
def get_product(lst):  
    p=1  
    for i in lst:  
        p*=i  
    print(p)  
  
get_product([1,2,3,4,5])
```



Output:

```
In [11]: runfile('C:/Users/rooman/OneDrive/Desktop/python/  
test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')  
1728000
```

Closures

Though we have seen the inner/outer function there is another tricky thing to know. Let us see what it is

```
def outer():  
    x=99  
  
    def inner():  
        print(x)  
    return inner  
  
fun=outer()  
fun()
```



Output:

```
In [12]: runfile('C:/Users/rooman/OneDrive/Desktop/python/  
test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')  
99
```

Now we are trying to delete the outer function. Let us see what happens

```
def outer():
    x=99

    def inner():
        print(x)
    return inner

fun=outer()
del outer
fun()
# will get the output because of concept called as closures
```

Output:

```
In [13]: runfile('C:/Users/rooman/OneDrive/Desktop/python/
test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
99
```

We get the output even after deleting the outer function because of the concept called as closures.

When the outer function is deleted and the inner function is still present, though the variables in outer function got deleted the inner function will still have the access to those variables, because somewhere in memory the values are still present.

Let us look at another example to understand correctly

```
def outer():
    x=99

    def inner1():
        y=88

        def inner2():
            print(x)
            print(y)
        return inner2
    return inner1

fun1=outer()
fun2=fun1()
fun2()
```



Output:

```
In [14]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
99
88
```

Now let us try deleting the outer functions

```
def outer():
    x=99

    def inner1():
        y=88

        def inner2():
            print(x)
            print(y)
        return inner2
    return inner1

fun1=outer()
fun2=fun1()
fun2()
del outer
del fun1
fun2()
```



Output:

```
In [15]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
99
88
99
88
```



Python Fundamentals

day 49

Today's Agenda

- Types of methods within class
- Instance method
- Static method
- Class method



Types of methods within class

The functions present within the class are called as methods. We can have 3 types of methods within the class

- ❖ Instance method
- ❖ Static method
- ❖ Class method

So far the methods we have come across are instance methods.

Let us take the example of BMW car and modify it according to the necessity and explore the different classes.

Instance methods

This is a very basic and easy method that we use regularly when we create classes in python. If we want to print an instance variable or instance method we must create an object of that required class.

If we are using `self` as a function parameter or in front of a variable, that is nothing but the calling instance itself.

As we are working with instance variables we use `self` keyword.

Note: Instance variables are used with instance methods.

```
class BmwCar:

    def __init__(self, name, cc, color):
        self.name=name
        self.cc=cc
        self.color=color

    def start_engine(self):
        print(self.name, 'engine is starting')

def main():
    c=BmwCar('bmw', 2100, 'blue')
    c.start_engine()
    #BmwCar.start_engine(c)

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



Output:

```
In [16]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
bmw engine is starting
```

Now let us modify the above code by adding function that can convert kilometres to miles.

```

class BmwCar:

    def __init__(self, name, cc, color):
        self.name=name
        self.cc=cc
        self.color=color

    def start_engine(self):
        print(self.name,'engine is starting')

    def kms_to_miles(kms):
        print(kms*1.6)

def main():
    c=BmwCar('bmw',2100,'blue')
    c.start_engine()
#BmwCar.start_engine(c)
BmwCar.kms_to_miles(2)

if __name__ == '__main__':
    main()

```



Output:

```

In [17]: runfile('C:/Users/rooman/OneDrive/Desktop/python/
test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
bmw engine is starting
3.2

```

Great! Using the class it successfully works. But can we do the same using the object? Let us see

```

class BmwCar:

    def __init__(self, name, cc, color):
        self.name=name
        self.cc=cc
        self.color=color

    def start_engine(self):
        print(self.name,'engine is starting')

    def kms_to_miles(kms):
        print(kms*1.6)

def main():
    c=BmwCar('bmw',2100,'blue')
    c.start_engine()
#BmwCar.start_engine(c)
BmwCar.kms_to_miles(2)
c.kms_to_miles(2)
#c.kms_to_miles(c,2)

if __name__ == '__main__':
    main()

```



Output:

```
File "C:/Users/rooman/OneDrive/Desktop/python/test.py",
line 19, in main
    c.kms_to_miles(2)

TypeError: kms_to_miles() takes 1 positional argument but
2 were given
```

We certainly cannot, because internally it is taking only one parameter but two were given. So let us see how to overcome this



Static method

A static method can be called without an object for that class, using the class name directly. If you want to do something extra with a class we use static methods.

```
class BmwCar:

    def __init__(self, name, cc, color):
        self.name = name
        self.cc = cc
        self.color = color

    def start_engine(self):
        print(self.name, 'engine is starting')

    @staticmethod
    def kms_to_miles(kms):
        print(kms * 1.6)

def main():
    c = BmwCar('bmw', 2100, 'blue')
    c.start_engine()
    #BmwCar.start_engine(c)
    BmwCar.kms_to_miles(2)
    c.kms_to_miles(2)
    #c.kms_to_miles(2)

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



Output:

```
In [19]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
bmw engine is starting
3.2
3.2
```

Class method

There are two ways to create class methods in python:

1. Using `classmethod(function)`
2. Using `@classmethod` annotation

As we are working with `ClassMethod` we use the `cls` keyword. Class variables are used with class methods.

```
class BmwCar:

    def __init__(self, name, cc, color):
        self.name=name
        self.cc=cc
        self.color=color

    @classmethod
    def x1(cls):
        return cls('x1', 1998, 'blue')

    @classmethod
    def series5(cls):
        return cls('5series', 2993, 'blue')

    @classmethod
    def i8(cls):
        return cls('i8', 1499, 'blue')

    def display(self):
        print(self.name)
        print(self.cc)
        print(self.color)

    def main():
        c1=BmwCar.x1()
        c2=BmwCar.series5()
        c3=BmwCar.i8()
        c1.display()
        c2.display()
        c3.display()

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



Output:

```
In [20]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
x1
1998
blue
5series
2993
blue
i8
1499
blue
```



Python fundamentals

day 50

Today's Agenda

- Design principals of object orientation



Design principals of object orientation

The four major principles of object orientation are:

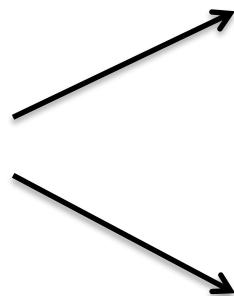
- Encapsulation
- Polymorphism
- Inheritance
- Abstraction

An object oriented program is based on classes and there exists a collection of interacting objects, as opposed to the conventional model, in which a program consists of functions and routines. In OOP, each object can receive messages, process data, and send messages to other objects.

Let us consider the same example of food delivery that we had considered earlier and develop it



Customer

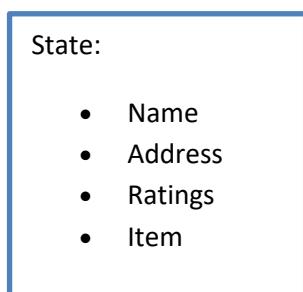


State:

- Name
- Email
- Phone_num
- Address

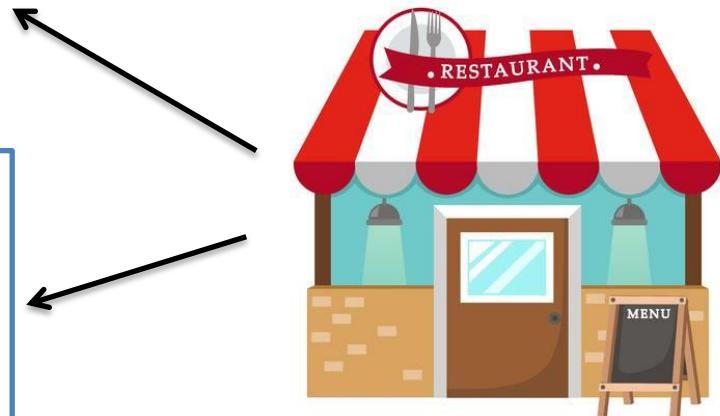
Behaviour:

- Sel_restaurant()
- Sel_item()
- Sel_address()
- Place_order()
- Cancel_order()
- Make_payment()
- Avail_discount()



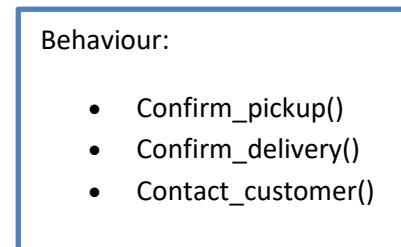
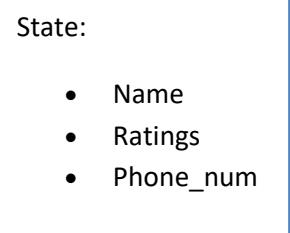
Behaviour:

- Confirm_order()
- Add_item()
- Del_item()

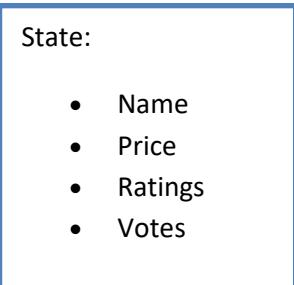




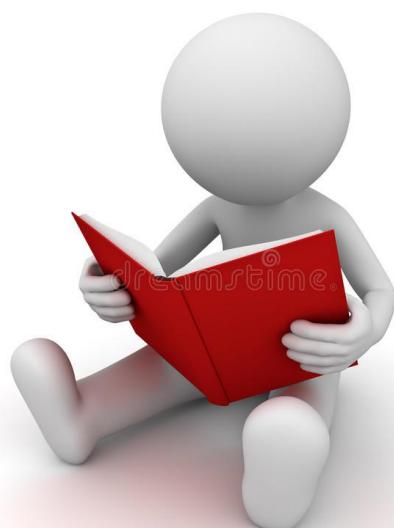
Delivery Agent

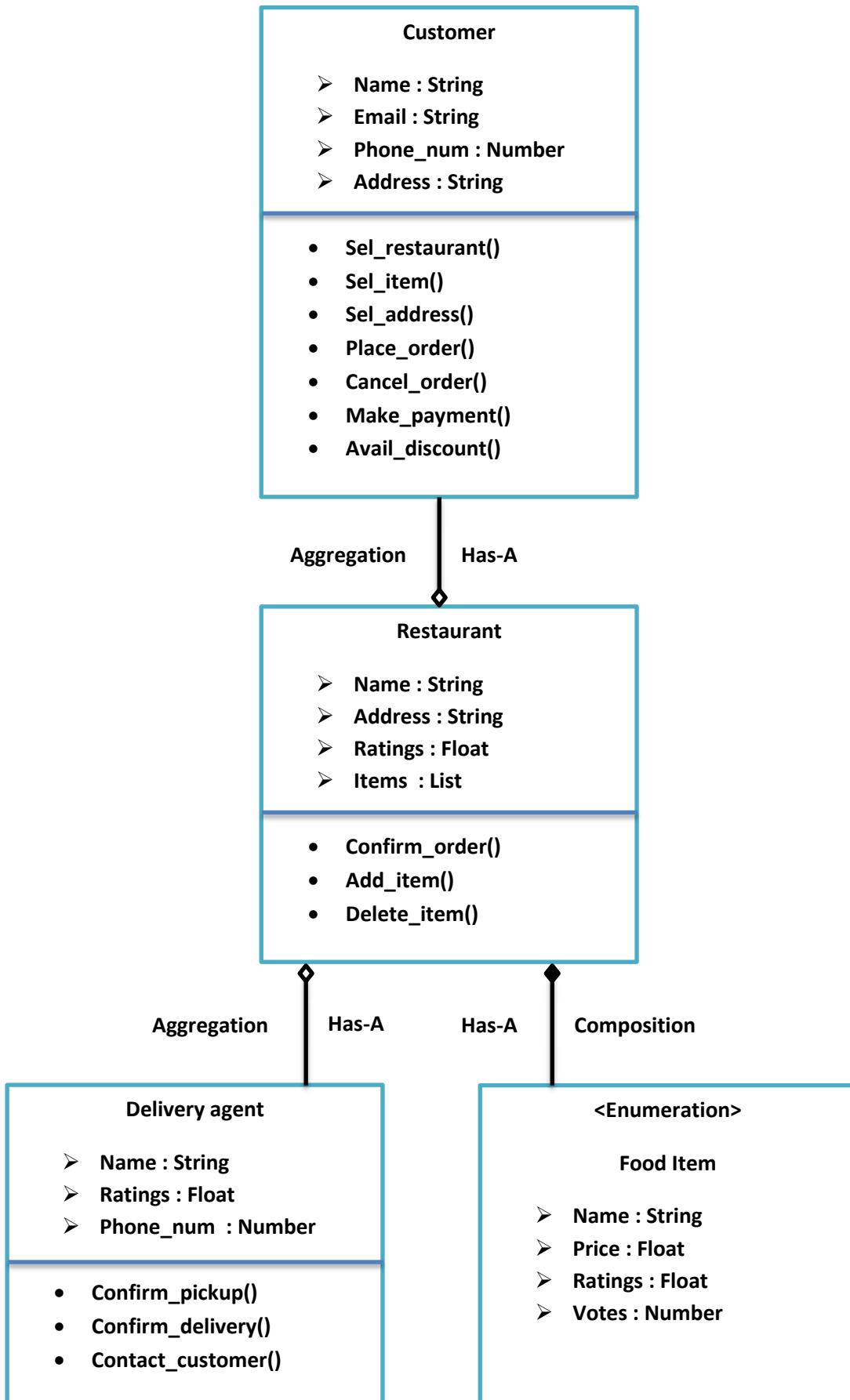


Food Item



Now all these objects should be having a connection between them.
Let us see





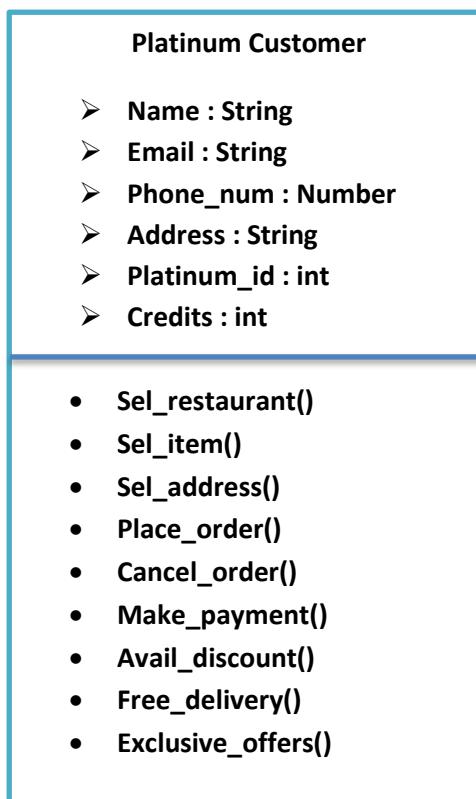
From the above UML we have derived the relation between all the objects. Now before going further let us see what is aggregation and composition

Composition and aggregation are specialised form of Association. Whereas Association is a relationship between two classes without any rules.

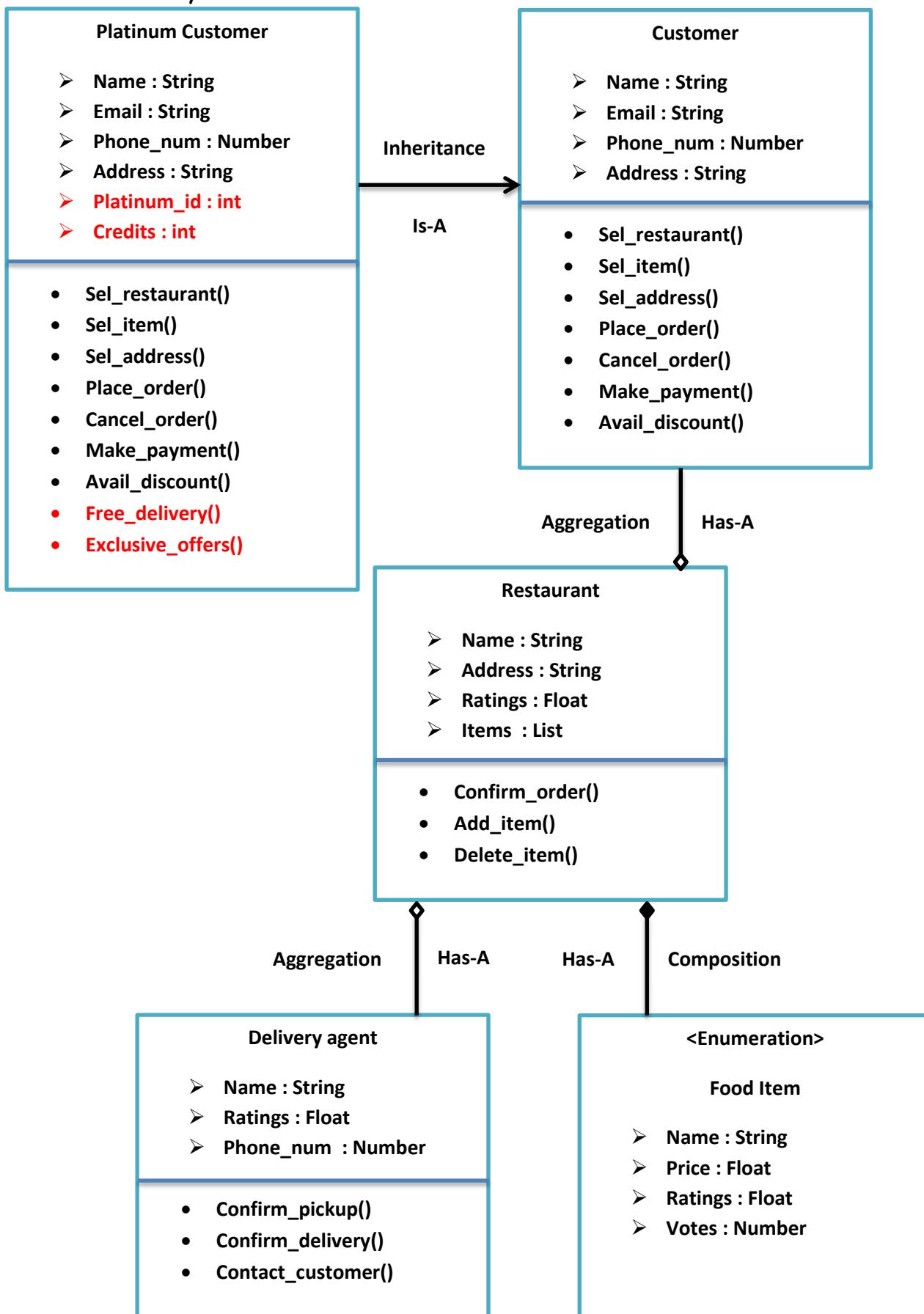
Composition: One class is container and other class is content and if you delete the container object then all of its contents objects are also deleted.

Aggregation: Aggregation is a weak form of composition. If you delete the container object contents objects can live without container object.

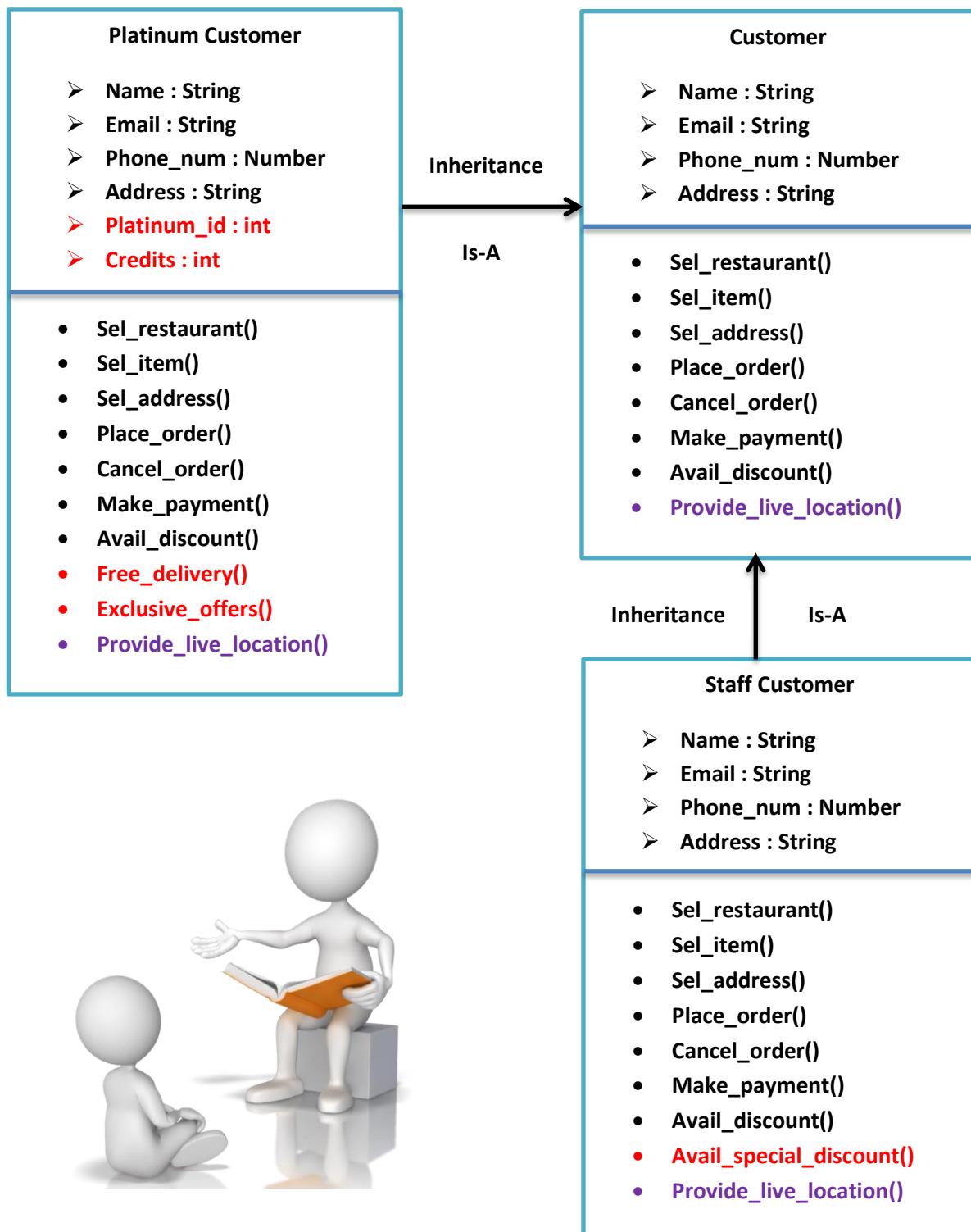
Now what if we want to add another feature and upgrade. It wouldn't be efficient way to start from the scratch. Therefore we have a feature where we can re-use the code and modify. Let us consider following as the new feature



Let us try to establish a relation between all the features



Now again we want to upgrade a feature where staff members are benefitted



Note: Any changes made in parent class reflect in child class as well.

Python Fundamentals

day 51

Today's Agenda

- Encapsulation



Encapsulation

To know what is encapsulation and is it needed. Let us consider an example

```
class AccountHolder:  
  
    def __init__(self):  
        self.bal=10000  
  
ah=AccountHolder()  
print(ah.bal)  
ah.bal=20000  
print(ah.bal)
```

Output:

```
In [2]: runfile('C:/Users/rooman/OneDrive/Desktop/python/  
test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')  
10000  
20000
```

Let us now try giving negative number and see if the value modifies

```
class AccountHolder:  
  
    def __init__(self):  
        self.bal=10000  
  
ah=AccountHolder()  
print(ah.bal)  
ah.bal=-20000  
print(ah.bal)
```

Output:

```
In [3]: runfile('C:/Users/rooman/OneDrive/Desktop/python/  
test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')  
10000  
-20000
```

But balance in ones' account is personal and should not be easily accessible by the third person. And more importantly we shouldn't be allowed to mishandle the data. Let us see how to resolve this issue

```
class AccountHolder:  
  
    def __init__(self):  
        self.bal=10000  
  
    def get_bal(self):  
        return self.bal  
  
    def set_bal(self,amt):  
        if amt>0:  
            self.bal=amt  
        else:  
            print('Invalid Amount')  
  
ah=AccountHolder()  
print(ah.get_bal())  
ah.set_bal(-20000)  
print(ah.get_bal())
```



Output:

```
In [11]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
10000
Invalid Amount
10000
```

Encapsulation in Python is the process of wrapping up variables and methods into a single entity. In programming, a class is an example that wraps all the variables and methods defined inside it.

Encapsulation acts as a protective layer by ensuring that, access to wrapped data is not possible by any code defined outside the class in which the wrapped data are defined. Encapsulation provides security by hiding the data from the outside world.

But in the above code we can still access amt by calling bal and can still modify. How to avoid it?

```
class AccountHolder:

    def __init__(self):
        self._bal=10000 # _ represents not to access directly
                         # or symbolises as private variable

    def get_bal(self):
        return self._bal

    def set_bal(self,amt):
        if amt>0:
            self._bal=amt
        else:
            print('Invalid Amount')

ah=AccountHolder()
print(ah.get_bal())
ah.set_bal(-20000)
print(ah.get_bal())
```



Output:

```
In [12]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
10000
Invalid Amount
10000
```

Although the above declaration of variable is just indication of not using directly, which means any third member can still access it directly who isn't aware of it. Which says that python doesn't strictly enforce anything it's more of a responsibility of the user and developer. Is there a solution for this? There certainly is let us see

```
class AccountHolder:

    def __init__(self):
        self._bal=10000 # _ represents not to access directly
                         # or symbolises as private variable

    def get_bal(self):
        return self._bal

    def set_bal(self,amt):
        if amt>0:
            self._bal=amt
        else:
            print('Invalid Amount')

ah=AccountHolder()
print(ah.__dict__) #checks if bal is stored in dict
```

Output:

```
In [1]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
{'_bal': 10000}
```

We confirmed that bal is stored in dictionary, but still we haven't resolved the situation completely

```

class AccountHolder:

    def __init__(self):
        self.__bal=10000

    def get_bal(self):
        return self.__bal

    def set_bal(self,amt):
        if amt>0:
            self.__bal=amt
        else:
            print('Invalid Amount')

ah=AccountHolder()
print(ah.__dict__) #checks if bal is stored in dict

```

Output:

```
In [2]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
{'_AccountHolder__bal': 10000}
```

By adding 2 underscore in front of the variable made the difference. Python internally changed the `__bal` to `_AccountHolder__bal` using a concept called as **data mangling**.

There is no strict way to stop the access there are only few tricks which we can work around with.

Now let us see another example of data mangling

```

class AccountHolder:

    def __init__(self):
        self.__bal=10000

    def get_bal(self):
        return self.__bal

    def set_bal(self,amt):
        if amt>0:
            self.__bal=amt
        else:
            print('Invalid Amount')

ah=AccountHolder()
ah.__bal=20000
print(ah.__dict__)

```



Output:

```
In [3]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
{'_AccountHolder__bal': 10000, '__bal': 20000}
```

We see that new variable is created with name `__bal`. This is because mangling is a process which is only applied to instance variables which are created within the class.

This is how internally new variable got created

```
class AccountHolder:

    def __init__(self):
        self.__bal=10000

    def get_bal(self):
        return self.__bal

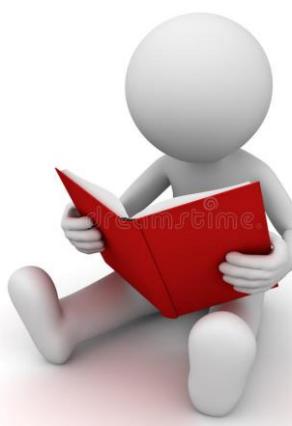
    def set_bal(self,amt):
        if amt>0:
            self.__bal=amt
        else:
            print('Invalid Amount')

ah=AccountHolder()
ah.__bal=20000 #ah.__dict['__bal']=20000
print(ah.__dict__)
```



The above code will give the same output.

Let us see another example



```

class AccountHolder:

    def __init__(self):
        self.__bal=10000

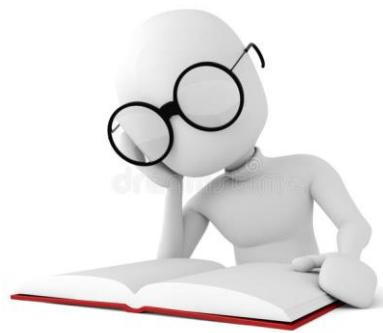
    def get_bal(self):
        print('get_bal() called')
        return self.__bal

    def set_bal(self,amt):
        print('set_bal called')
        if amt>0:
            self.__bal=amt
        else:
            print('Invalid Amount')

    bal=property(get_bal,set_bal)

ah=AccountHolder()
print(ah.bal)
ah.bal=20000
print(ah.bal)

```



There's another way to do the same.

```

class AccountHolder:

    def __init__(self):
        self.__bal=10000

    @property
    def bal(self):
        print('get_bal() called')
        return self.__bal

    @bal.setter
    def bal(self,amt):
        print('set_bal called')
        if amt>0:
            self.__bal=amt
        else:
            print('Invalid Amount')

    #bal=property(get_bal,set_bal)

ah=AccountHolder()
print(ah.bal)
ah.bal=20000
print(ah.bal)

```



Output:

```

In [6]: runfile('C:/Users/rooman/OneDrive/Desktop/python/
test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
get_bal() called
10000
set_bal called
get_bal() called
20000

```

Python Fundamentals

day 53

Today's Agenda

- super() in inheritance



super() in Inheritance

The **super** keyword refers to superclass (parent) objects. It is used to call superclass methods, and to access the superclass constructor. The most common use of the **super** keyword is to eliminate the confusion between super-classes and sub-classes that have methods with the same name.

Let us look at an example and see how to use
super()



```

class Customer:

    def __init__(self, name, ph, email):
        self.name=name
        self.ph=ph
        self.email=email

class PlatinumCustomer(Customer):
    def __init__(self, name, ph, email, plat_id):
        self.name=name
        self.ph=ph
        self.email=email
        self.plat_id=plat_id
    def display(self):
        print(self.__dict__)

def main():
    p=PlatinumCustomer('Rohit', 9900887766, 'rohit@gmail.com', 10)
    p.display()

if __name__=='__main__':
    main()

```

Output:

```

In [1]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
{'name': 'Rohit', 'ph': 9900887766, 'email': 'rohit@gmail.com', 'plat_id': 10}

```

In the above example we can reduce the size of `__init__()` of child class i.e. `PlatinumCustomer()` by using `super()`. Let us see below how to do it



```

class Customer:

    def __init__(self, name, ph, email):
        self.name=name
        self.ph=ph
        self.email=email

class PlatinumCustomer(Customer):
    def __init__(self, name, ph, email, plat_id):
        super().__init__(name, ph, email)
        self.plat_id=plat_id
    def display(self):
        print(self.__dict__)

def main():
    p=PlatinumCustomer('Rohit', 9900887766, 'rohit@gmail.com', 10)
    p.display()

if __name__=='__main__':
    main()

```

Output:

```

In [2]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
{'name': 'Rohit', 'ph': 9900887766, 'email': 'rohit@gmail.com', 'plat_id': 10}

```

We can also call any other method of parent class using `super()`. Let us take another example to know how



```

class Customer:

    def __init__(self, name, addr, ph_no):
        self.name = name
        self.addr = addr
        self.ph_no = ph_no

    def place_order(self, dish):
        cost = 0
        del_charge = 50
        if dish == 'pizza':
            cost = 500 + del_charge
        else:
            cost = 250 + del_charge
        return cost

class PlatinumCustomer(Customer):

    def __init__(self, name, addr, ph_no, plat_id):
        super().__init__(name, addr, ph_no)
        self.plat_id = plat_id

    def place_order(self, dish):
        del_charge = 50
        return (super().place_order(dish) - del_charge) * 0.95

def main():
    p = PlatinumCustomer('Rohit', 'ITC', 9900887766, 12)
    print(p.place_order('pizza'))

if __name__ == '__main__':
    main()

```



Output:

```
In [5]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
475.0
```

Next let us see how to use super() in multilevel inheritance

```

class A:
    def fun(self):
        print('A')

class B(A):
    def fun(self):
        print('B')

class C(B):
    def fun(self):
        super().fun() #super(C, self).fun()
        print('C')
c=C()
c.fun()

```

Output:

```
In [6]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
B
C
```

Great!! But can we access class A methods as well? Certainly we can

```
class A:
    def fun(self):
        print('A')

class B(A):
    def fun(self):
        print('B')

class C(B):
    def fun(self):
        super(B,self).fun()
        print('C')
c=C()
c.fun()
```

Output:

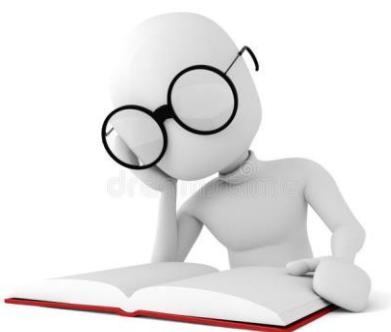
```
In [7]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
A
C
```

Now let us go ahead to multiple inheritance

```
class A:
    def fun(self):
        print('A')

class B:
    def fun(self):
        print('B')

class C(A,B):
    def test(self):
        super().fun()
        print('C')
c=C()
#help(c)
c.test()
```



Output:

```
In [10]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
A
C
```

What if we want to check whether the object or variable is an instance of specified class type or datatype and check if a class is subclass of another class. How do we do it?



```
class A:
    def fun(self):
        print('A')

class B:
    def fun(self):
        print('B')

class C(A,B):
    def test(self):
        super().fun()
        print('C')
c=C()
#help(C)
#c.test()
print(isinstance(c,int))
print(issubclass(C,object))
```

Output:

```
In [12]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
False
True
```

Python Fundamentals

day 53

Today's Agenda

- super() in inheritance



super() in Inheritance

The **super** keyword refers to superclass (parent) objects. It is used to call superclass methods, and to access the superclass constructor. The most common use of the **super** keyword is to eliminate the confusion between super-classes and sub-classes that have methods with the same name.

Let us look at an example and see how to use
super()



```

class Customer:

    def __init__(self, name, ph, email):
        self.name=name
        self.ph=ph
        self.email=email

class PlatinumCustomer(Customer):
    def __init__(self, name, ph, email, plat_id):
        self.name=name
        self.ph=ph
        self.email=email
        self.plat_id=plat_id
    def display(self):
        print(self.__dict__)

def main():
    p=PlatinumCustomer('Rohit', 9900887766, 'rohit@gmail.com', 10)
    p.display()

if __name__=='__main__':
    main()

```

Output:

```

In [1]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
{'name': 'Rohit', 'ph': 9900887766, 'email': 'rohit@gmail.com', 'plat_id': 10}

```

In the above example we can reduce the size of `__init__()` of child class i.e. `PlatinumCustomer()` by using `super()`. Let us see below how to do it



```

class Customer:

    def __init__(self, name, ph, email):
        self.name=name
        self.ph=ph
        self.email=email

class PlatinumCustomer(Customer):
    def __init__(self, name, ph, email, plat_id):
        super().__init__(name, ph, email)
        self.plat_id=plat_id
    def display(self):
        print(self.__dict__)

def main():
    p=PlatinumCustomer('Rohit', 9900887766, 'rohit@gmail.com', 10)
    p.display()

if __name__=='__main__':
    main()

```

Output:

```

In [2]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
{'name': 'Rohit', 'ph': 9900887766, 'email': 'rohit@gmail.com', 'plat_id': 10}

```

We can also call any other method of parent class using `super()`. Let us take another example to know how



```

class Customer:

    def __init__(self, name, addr, ph_no):
        self.name = name
        self.addr = addr
        self.ph_no = ph_no

    def place_order(self, dish):
        cost = 0
        del_charge = 50
        if dish == 'pizza':
            cost = 500 + del_charge
        else:
            cost = 250 + del_charge
        return cost

class PlatinumCustomer(Customer):

    def __init__(self, name, addr, ph_no, plat_id):
        super().__init__(name, addr, ph_no)
        self.plat_id = plat_id

    def place_order(self, dish):
        del_charge = 50
        return (super().place_order(dish) - del_charge) * 0.95

def main():
    p = PlatinumCustomer('Rohit', 'ITC', 9900887766, 12)
    print(p.place_order('pizza'))

if __name__ == '__main__':
    main()

```



Output:

```
In [5]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
475.0
```

Next let us see how to use super() in multilevel inheritance

```

class A:
    def fun(self):
        print('A')

class B(A):
    def fun(self):
        print('B')

class C(B):
    def fun(self):
        super().fun() #super(C, self).fun()
        print('C')
c=C()
c.fun()

```

Output:

```
In [6]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
B
C
```

Great!! But can we access class A methods as well? Certainly we can

```
class A:
    def fun(self):
        print('A')

class B(A):
    def fun(self):
        print('B')

class C(B):
    def fun(self):
        super(B,self).fun()
        print('C')
c=C()
c.fun()
```

Output:

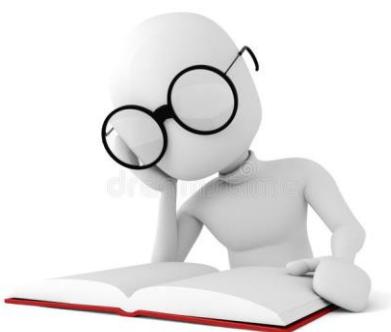
```
In [7]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
A
C
```

Now let us go ahead to multiple inheritance

```
class A:
    def fun(self):
        print('A')

class B:
    def fun(self):
        print('B')

class C(A,B):
    def test(self):
        super().fun()
        print('C')
c=C()
#help(c)
c.test()
```



Output:

```
In [10]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
A
C
```

What if we want to check whether the object or variable is an instance of specified class type or datatype and check if a class is subclass of another class. How do we do it?



```
class A:
    def fun(self):
        print('A')

class B:
    def fun(self):
        print('B')

class C(A,B):
    def test(self):
        super().fun()
        print('C')
c=C()
#help(C)
#c.test()
print(isinstance(c,int))
print(issubclass(C,object))
```

Output:

```
In [12]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
False
True
```

Python Fundamentals

day 54

Today's Agenda

- Extending built-in classes



Extending built-in classes

We can extend all of Python's built-in classes. This allows us to add or modify features of the data types that come with Python. This may save us from having to build a program from scratch. Let us take an example and see how to achieve it

```
class Contact:  
    all_contacts=[]  
  
    def __init__(self, name, email):  
        self.name=name  
        self.email=email  
        Contact.all_contacts.append(self)  
  
    def display(self):  
        print(self.__dict__)  
  
def main():  
    c1=Contact('Rohit','rohit@gmail.com')  
    c2=Contact('Manish','mainsh@gmail.com')  
    print(Contact.all_contacts)  
  
if __name__=='__main__':  
    main()
```



Output:

```
In [2]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
[<__main__.Contact object at 0x000001AFF32E1088>,
 <__main__.Contact object at 0x000001AFF327D3C8>]
```

We have list of two objects. Let us try to print the instances within those objects

```
class Contact:
    all_contacts=[]

    def __init__(self, name, email):
        self.name=name
        self.email=email
        Contact.all_contacts.append(self)

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

def main():
    c1=Contact('Rohit', 'rohit@gmail.com')
    c2=Contact('Manish', 'mainsh@gmail.com')
    for i in Contact.all_contacts:
        i.display()

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



Output:

```
In [3]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
{'name': 'Rohit', 'email': 'rohit@gmail.com'}
{'name': 'Manish', 'email': 'mainsh@gmail.com'}
```

It would be a lot easier if we could get the above output by calling a function as shown below in the comment section.

```

class Contact:
    all_contacts=[]

    def __init__(self,name,email):
        self.name=name
        self.email=email
        Contact.all_contacts.append(self)

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

def main():
    c1=Contact('Rohit','rohit@gmail.com')
    c2=Contact('Manish','mainsh@gmail.com')
    for i in Contact.all_contacts:
        i.display()
    #Contact.all_contacts.display_all_contacts()

    name='Rohit'
    for i in Contact.all_contacts:
        if i.name==name:
            print('contact found')
    #Contact.all_contacts.search_contact('Rohit')

if __name__=='__main__':
    main()

```

Output:

```

In [5]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
{'name': 'Rohit', 'email': 'rohit@gmail.com'}
{'name': 'Manish', 'email': 'mainsh@gmail.com'}
contact found

```

Let us now see how to extend the features of built-in data types using class in the below example



```

class ContactList(list):
    def display_all_contacts(self):
        for i in self:
            i.display()

    def search_contact(self, name):
        for i in self:
            if i.name==name:
                return 'Contact found'
        return 'Contact not found'

class Contact:
    all_contacts=ContactList()

    def __init__(self, name, email):
        self.name=name
        self.email=email
        Contact.all_contacts.append(self)

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

def main():
    c1=Contact('Rohit', 'rohit@gmail.com')
    c2=Contact('Manish', 'mainsh@gmail.com')

    Contact.all_contacts.display_all_contacts()

    print(Contact.all_contacts.search_contact('Rohit'))

if __name__=='__main__':
    main()

```



Output:

```

In [8]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
{'name': 'Rohit', 'email': 'rohit@gmail.com'}
{'name': 'Manish', 'email': 'mainsh@gmail.com'}
Contact found

```

All we did is, instead of directly creating a list we created an object of the ContactList which is a list on which we can activate the extended features display_all_contacts and search_contacts which a traditional list did not have.

Python Fundamentals

day 55

Today's Agenda

- Polymorphism
- Duck Typing



Polymorphism

Polymorphism in python defines methods in the child class that have the same name as the methods in the parent class. In inheritance, the child class inherits the methods from the parent class. Also, it is possible to modify a method in a child class that it has inherited from the parent class.

Polymorphism establishes 1:many relation. This means certain line of code giving different output, or performing different task.

Let us look at the below example which is non-polymorphic and then convert the same into a code following polymorphism



```

class Messenger:
    def use_keyboard(self):
        print('Using Keyboard')
    def send_message(self):
        print('Text message sent')
    def receive_message(self):
        print('text message received')

class WhatsApp(Messenger):
    def send_message(self):
        print('Text, video and audio message sent using WA')
    def receive_message(self):
        print('text, video and audio message received using WA')

class FacebookMessenger(Messenger):
    def send_message(self):
        print('Text, video and audio message sent using FBM')
    def receive_message(self):
        print('text, video and audio message received using FBM')

class InstaMessenger(Messenger):
    def send_message(self):
        print('Text, video and audio message sent using Insta')
    def receive_message(self):
        print('text, video and audio message received using Insta')

wa=WhatsApp()
fb=FacebookMessenger()
im=InstaMessenger()

wa.use_keyboard()
wa.send_message()
wa.receive_message()

fb.use_keyboard()
fb.send_message()
fb.receive_message()

im.use_keyboard()
im.send_message()
im.receive_message()

```



Output:

```

In [1]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
Using Keyboard
Text, video and audio message sent using WA
text, video and audio message received using WA
Using Keyboard
Text, video and audio message sent using FBM
text, video and audio message received using FBM
Using Keyboard
Text, video and audio message sent using Insta
text, video and audio message received using Insta

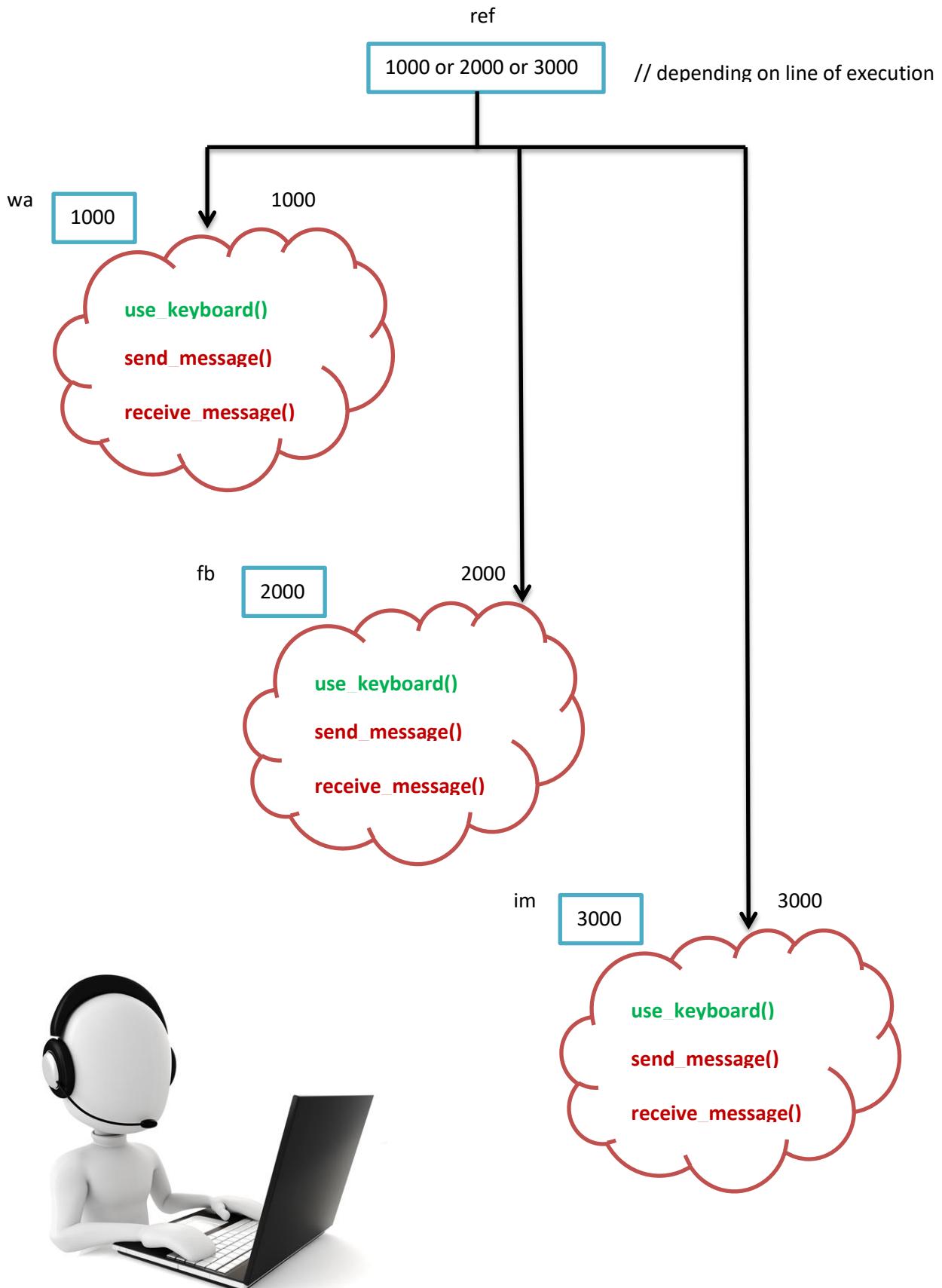
```

Now let us write the code implementing polymorphism, to achieve code flexibility and code reusability.

```
class Messenger:  
    def use_keyboard(self):  
        print('Using Keyboard')  
    def send_message(self):  
        print('Text message sent')  
    def receive_message(self):  
        print('text message received')  
  
class WhatsApp(Messenger):  
    def send_message(self):  
        print('Text, video and audio message sent')  
    def receive_message(self):  
        print('text, video and audio message received')  
  
class FacebookMessenger(Messenger):  
    def send_message(self):  
        print('Text, video and audio message sent')  
    def receive_message(self):  
        print('text, video and audio message received')  
  
class InstaMessenger(Messenger):  
    def send_message(self):  
        print('Text, video and audio message sent')  
    def receive_message(self):  
        print('text, video and audio message received')  
  
def use_messenger(ref):  
    ref.use_keyboard()  
    ref.send_message()  
    ref.receive_message()  
  
wa=WhatsApp()  
fb=FacebookMessenger()  
im=InstaMessenger()  
use_messenger(wa)  
use_messenger(fb)  
use_messenger(im)
```



Let us trace before seeing the output



Output:

```
In [2]: runfile('C:/Users/rooman/OneDrive/Desktop/pyt
test.py', wdir='C:/Users/rooman/OneDrive/Desktop/pyt
Using Keyboard
Text, video and audio message sent using WA
text, video and audio message received using WA
Using Keyboard
Text, video and audio message sent using FBM
text, video and audio message received using FBM
Using Keyboard
Text, video and audio message sent using Insta
text, video and audio message received using Insta
```

Great! But what if we want to include specialized methods? Duck typing is the solution. Didn't understand? Let us explore

Duck typing



In duck typing, you do not check types at all. Instead, you check for the presence of a given method or attribute. Like shown in the below example

```
class Messenger:
    def use_keyboard(self):
        print('Using Keyboard')
    def send_message(self):
        print('Text message sent')
    def receive_message(self):
        print('text message received')

class WhatsApp(Messenger):
    def send_message(self):
        print('Text, video and audio message sent using WA')
    def receive_message(self):
        print('text, video and audio message received using WA')
    def send_live_location(self):
        print('Live Location sent using WA')

class FacebookMessenger(Messenger):
    def send_message(self):
        print('Text, video and audio message sent using FBM')
    def receive_message(self):
        print('text, video and audio message received using FBM')
    def use_builtin_apps(self):
        print('using built-in apps using FBM')
```

```

class InstaMessenger(Messenger):
    def send_message(self):
        print('Text, video and audio message sent using Insta')
    def receive_message(self):
        print('text, video and audio message received using Insta')
    def add_filters(self):
        print('Filters using insta')

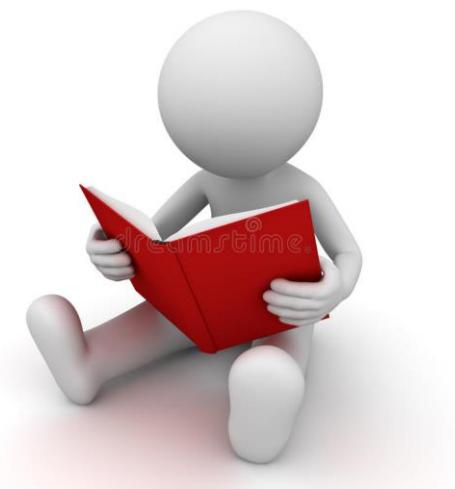
def use_messenger(ref):
    ref.use_keyboard()
    ref.send_message()
    ref.receive_message()

    if type(ref)== WhatsApp:
        ref.send_live_location()
    if type(ref)==FacebookMessenger:
        ref.use_builtin_apps()
    if type(ref)==InstaMessenger:
        ref.add_filters()

wa=WhatsApp()
fb=FacebookMessenger()
im=InstaMessenger()

use_messenger(wa)
use_messenger(fb)
use_messenger(im)

```



Output:

```

In [3]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
Using Keyboard
Text, video and audio message sent using WA
text, video and audio message received using WA
Live Location sent using WA
Using Keyboard
Text, video and audio message sent using FBM
text, video and audio message received using FBM
using built-in apps using FBM
Using Keyboard
Text, video and audio message sent using Insta
text, video and audio message received using Insta
Filters using insta

```

Great! So now we know how to make code flexible with specialized methods as well.

Python Fundamentals

day 56

Today's Agenda

- Magic methods
- Operator overloading



Magic Methods

Dunder or magic methods in Python are the methods having two prefix and suffix underscores in the method name. Dunder here means "Double Under (Underscores)". These are commonly used for operator overloading. Few examples for magic methods are:

`__init__`, `__add__`, `__len__`, `__repr__` etc.

Let us take an example and see how operator overloading is done

```
def main():
    a=5
    print(a)
    a=a+5 #a=a.__add__(5)
    print(a)
if __name__=='__main__':
    main()
```

Output:

```
In [1]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
5
10
```

In the above example `a=a+5` executes in the following manner

```
class int:
    .
    .
    def __add__(self,other):
        .
        .
        .
        return self + other
```

Similar to the above operation subtraction, multiplication, division and power also execute in same manner.

`a=a-5 → a=a.__sub__(5)`

`a=a*5 → a=a.__mul__(5)`

`a=a/5 → a=a.__div__(5)`

`a=a**5 → a=a.__pow__(5)`

So let us cross verify by using one of these magic method in the above code.

```
def main():
    a=5
    print(a)
    a=a.__add__(5)
    print(a)
if __name__=='__main__':
    main()
```

Output:

```
In [2]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
5
10
```

Let us next take example of concatenation and see how it works, because we use the same + operator for concatenation as well

```
def main():
    s='pyt'
    print(s)
    s=s+'hon' #s=s.__add__('hon')
    print(s)
if __name__=='__main__':
    main()
```

Output:

```
In [3]: runfile('C:/Users/rooman/OneDrive/Desktop/python/
test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
pyt
python
```

Note that here string class is executing not the int class.

```
class str:
    .
    .
    .
    def __add__(self,other):
        .
        .
        .
        .
```

If we try to mix both integer and string and then use + operator, you will definitely get an error as shown below

```
def main():
    s='pyt'
    print(s)
    s=s+5 #s=s.__add__('5')
    print(s)
if __name__=='__main__':
    main()
```



Output:

```
File "C:/Users/rooman/OneDrive/Desktop/python/test.py",
line 7, in <module>
    main()

File "C:/Users/rooman/OneDrive/Desktop/python/test.py",
line 4, in main
    s=s+5 #s=s.__add__('5')

TypeError: can only concatenate str (not "int") to str
```

If we reverse the order then as well we will get the error

```
def main():
    s='pyt'
    print(s)
    s=5+s #s=5.__add__('s')
    print(s)
if __name__=='__main__':
    main()
```



Output:

```
File "C:/Users/rooman/OneDrive/Desktop/python/test.py",
line 7, in <module>
    main()

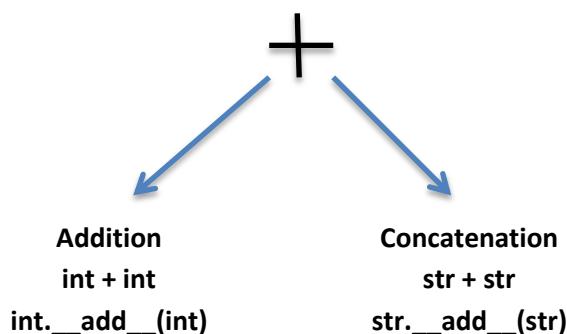
File "C:/Users/rooman/OneDrive/Desktop/python/test.py",
line 4, in main
    s=5+s #s=5.__add__('s')

TypeError: unsupported operand type(s) for +: 'int' and
'str'
```

So be careful when using operators and do not mix the different types of data

Operator overloading

Let us take an example to understand what is operator overloading. In the above example we saw + operator will perform arithmetic addition on two numbers, merge two lists, or concatenate two strings. This feature in Python that allows the same operator to have different meaning according to the context is called operator overloading.



All the above examples are on built-in methods. Let us see how to achieve the same with user defined type.

```
class Point:  
  
    def __init__(self,x,y):  
        self.x=x  
        self.y=y  
  
    def display(self):  
        print(self.__dict__)  
  
def main():  
    p1=Point(2,3)  
    p2=Point(1,1)  
    p1.display()  
    p2.display()  
  
if __name__=='__main__':  
    main()
```



Output:

```
In [6]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
{'x': 2, 'y': 3}
{'x': 1, 'y': 1}
```

Now let us see if we can get the third point which will be the added result of these two points.

```
class Point:

    def __init__(self,x,y):
        self.x=x
        self.y=y

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

    def main():
        p1=Point(2,3)
        p2=Point(1,1)
        p3=p1+p2 #p3=p1.__add__(p2)

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



Output:

```
File "C:/Users/rooman/OneDrive/Desktop/python/test.py",
line 16, in <module>
  main()

  File "C:/Users/rooman/OneDrive/Desktop/python/test.py",
line 13, in main
  p3=p1+p2

TypeError: unsupported operand type(s) for +: 'Point' and
'Point'
```

Certainly above code will throw error because there is no `__add__` method in point class. So let us see how to create a `__add__` method and see if it works

```

class Point:

    def __init__(self,x,y):
        self.x=x
        self.y=y

    def __add__(self,other):
        return Point(self.x + other.x, self.y + other.y)

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

def main():
    p1=Point(2,3)
    p2=Point(1,1)
    p3=p1+p2 #p3=p1.__add__(p2)
    p3.display()

if __name__=='__main__':
    main()

```

Output:

```

In [8]: runfile('C:/Users/rooman/OneDrive/Desktop/python/
test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
{'x': 3, 'y': 4}

```

Let us see if we can print p1, p2, p3 using print() and not by the display().

```

class Point:

    def __init__(self,x,y):
        self.x=x
        self.y=y

    def __add__(self,other):
        return Point(self.x + other.x, self.y + other.y)

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

def main():
    p1=Point(2,3)
    p2=Point(1,1)
    p3=p1+p2 #p3=p1.__add__(p2)

    print(p1)
    print(p2)
    print(p3)

if __name__=='__main__':
    main()

```



Output:

```
In [10]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
<__main__.Point object at 0x0000027BA95EC9C8>
<__main__.Point object at 0x0000027BA95FB508>
<__main__.Point object at 0x0000027BA95FB548>
```

Instead of printing the values, it is printing the object. This is because of the absence of certain feature in `__str__()` because `print()` expects the arguments in it to be string type. If not it tries to convert it into string.

As `__str__()` is inherited by point class from object class, we can override it and modify according to our needs as shown below

```
class Point:

    def __init__(self,x,y):
        self.x=x
        self.y=y

    def __add__(self,other):
        return Point(self.x + other.x, self.y + other.y)

    def __str__(self):
        return f'({self.x},{self.y})'

def main():
    p1=Point(2,3)
    p2=Point(1,1)
    p3=p1+p2 #p3=p1.__add__(p2)

    print(p1.__str__())
    print(p2)
    print(p3)

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



Output:

```
In [12]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
(2,3)
(1,1)
(3,4)
```

But what if the point objects are identical as below and you are not sure if they are same objects or different objects. How to distinguish? Let us see

```
class Point:

    def __init__(self,x,y):
        self.x=x
        self.y=y

    def __add__(self,other):
        return Point(self.x + other.x, self.y + other.y)

    def __str__(self):
        return f'({self.x},{self.y})'

    def __repr__(self):
        return f'{type(self)} {id(self)}'

def main():
    p1=Point(2,3)
    p2=Point(1,1)

    print(p1)
    print(p2)
    print(p1.__repr__())
    print(p2.__repr__())

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



Output:

```
In [14]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
(2,3)
(1,1)
<class '__main__.Point'> 2730145810440
<class '__main__.Point'> 2730145283592
```

Great so now we know they are two different objects.

Python Fundamentals

day 57

Today's Agenda

- Has-A relationship



Has-A relationship

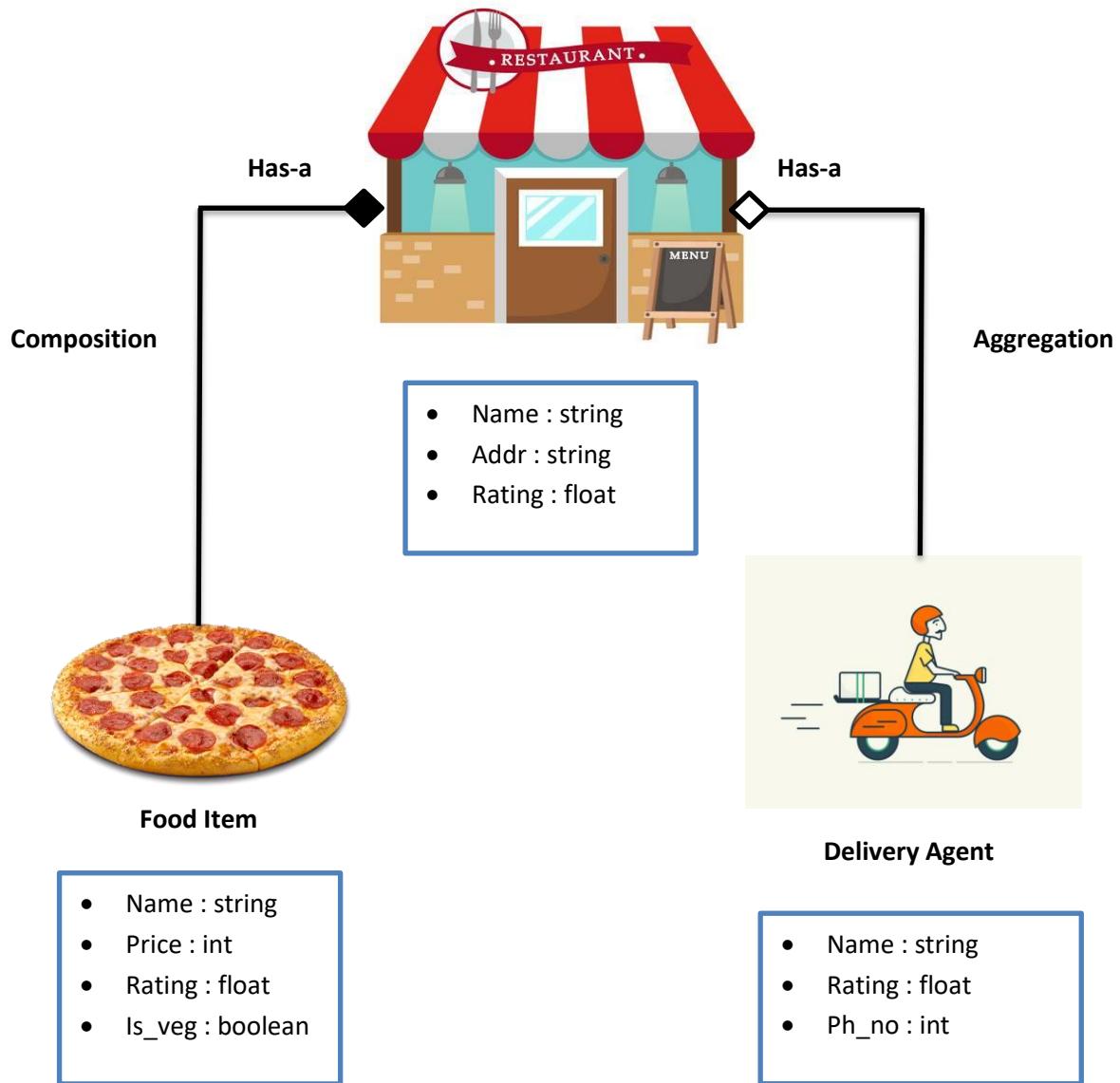
We have seen is-a relationship which can be achieved using inheritance but there is also has-a relationship which can be achieved using aggregation and composition.

Composition and aggregation are specialised form of Association. Where Association is a relationship between two classes without any rules.

In composition, one of the classes is composed of one or more instance of other classes. In other words, one class is container and other class is content and if you delete the container object then all of its contents objects are also deleted.

Aggregation is a weak form of composition. If you delete the container object contents objects can live without container object.

Let us take an example of restaurant and understand it correctly



Let us write the code for above
UML



```

class FoodItem:

    def __init__(self, name, price, rating, is_veg):
        self.name = name
        self.price = price
        self.rating = rating
        self.is_veg = is_veg

class DeliverAgent():

    def __init__(self, name, rating, ph_no):
        self.name = name
        self.rating = rating
        self.ph_no = ph_no

class Restaurant:

    def __init__(self, name, addr, rating):
        self.name = name
        self.addr = addr
        self.rating = rating
        self.pizza = FoodItem('Pizza', 500, 4.5, False)

    def assign_delivery_agent(self, agent):
        self.agent = agent

def main():
    r = Restaurant('XYZ', 'Bangalore', 4.7)
    steve = DeliverAgent('Steve', 4.2, 9900886655)
    r.assign_delivery_agent(steve)
    print(r.pizza.price)
    print(r.agent.name)

main()

```

Output:

```

In [2]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
500
Steve

```

Let us see if we can access food item and delivery agent without the restaurant



```

class FoodItem:

    def __init__(self, name, price, rating, is_veg):
        self.name = name
        self.price = price
        self.rating = rating
        self.is_veg = is_veg

class DeliverAgent():

    def __init__(self, name, rating, ph_no):
        self.name = name
        self.rating = rating
        self.ph_no = ph_no

class Restaurant:

    def __init__(self, name, addr, rating):
        self.name = name
        self.addr = addr
        self.rating = rating
        self.pizza = FoodItem('Pizza', 500, 4.5, False)

    def assign_delivery_agent(self, agent):
        self.agent = agent

def main():
    r = Restaurant('XYZ', 'Bangalore', 4.7)
    steve = DeliverAgent('Steve', 4.2, 9900886655)
    r.assign_delivery_agent(steve)
    print(r.pizza.price)
    print(r.agent.name)
    #print(pizza.price) throw error(tight bound-composition)
    print(steve.name) # Loose bound - aggregation

main()

```

Output:

```

In [4]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
500
Steve
Steve

```

Next let us see if we remove restaurant from the equation can we still access food and delivery agent



```

class FoodItem:

    def __init__(self, name, price, rating, is_veg):
        self.name = name
        self.price = price
        self.rating = rating
        self.is_veg = is_veg

class DeliverAgent():

    def __init__(self, name, rating, ph_no):
        self.name = name
        self.rating = rating
        self.ph_no = ph_no

class Restaurant:

    def __init__(self, name, addr, rating):
        self.name = name
        self.addr = addr
        self.rating = rating
        self.pizza = FoodItem('Pizza', 500, 4.5, False)

    def assign_delivery_agent(self, agent):
        self.agent = agent

def main():
    r = Restaurant('XYZ', 'Bangalore', 4.7)
    steve = DeliverAgent('Steve', 4.2, 9900886655)
    r.assign_delivery_agent(steve)
    print(r.pizza.price)
    print(r.agent.name)

    del r
    print(r.pizza.price)
    print(r.agent.name)
main()

```

Output:

```

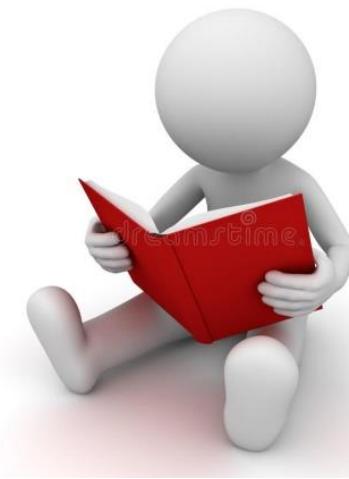
File "C:/Users/rooman/OneDrive/Desktop/python/test.py",
line 35, in main
    print(r.pizza.price)

```

```

UnboundLocalError: local variable 'r' referenced before
assignment

```



We cannot access food without restaurant which makes complete sense but we should be able to access delivery agent without restaurant

```
class FoodItem:

    def __init__(self, name, price, rating, is_veg):
        self.name = name
        self.price = price
        self.rating = rating
        self.is_veg = is_veg

class DeliverAgent():

    def __init__(self, name, rating, ph_no):
        self.name = name
        self.rating = rating
        self.ph_no = ph_no

class Restaurant:

    def __init__(self, name, addr, rating):
        self.name = name
        self.addr = addr
        self.rating = rating
        self.pizza = FoodItem('Pizza', 500, 4.5, False)

    def assign_delivery_agent(self, agent):
        self.agent = agent

def main():
    r = Restaurant('XYZ', 'Bangalore', 4.7)
    steve = DeliverAgent('Steve', 4.2, 9900886655)
    r.assign_delivery_agent(steve)
    print(r.pizza.price)
    print(r.agent.name)

    del r
    #print(r.pizza.price)
    print(steve.name)
main()
```



Output:

```
In [6]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
500
Steve
Steve
```

We see that Food item is outside the restaurant class, which means by creating an object of food item the instances can still be accessed. But that is not how things are in reality. Food items are present only inside restaurant. Let us see if we can implement the same even in the above code.

```
class DeliverAgent():

    def __init__(self, name, rating, ph_no):
        self.name = name
        self.rating = rating
        self.ph_no = ph_no

class Restaurant:

    class FoodItem: #innerclass

        def __init__(self, name, price, rating, is_veg):
            self.name = name
            self.price = price
            self.rating = rating
            self.is_veg = is_veg

        def __init__(self, name, addr, rating):
            self.name = name
            self.addr = addr
            self.rating = rating
            self.pizza = Restaurant.FoodItem('Pizza', 500, 4.5, False)

        def assign_delivery_agent(self, agent):
            self.agent = agent

    def main():
        r = Restaurant('XYZ', 'Bangalore', 4.7)
        steve = DeliverAgent('Steve', 4.2, 9900886655)
        r.assign_delivery_agent(steve)
        print(r.pizza.price)
        print(r.agent.name)

main()
```

Output:

```
In [7]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
500
Steve
```

Make sure whenever you are trying to access fooditem, first call restaurant.

Python Fundamentals

day 59

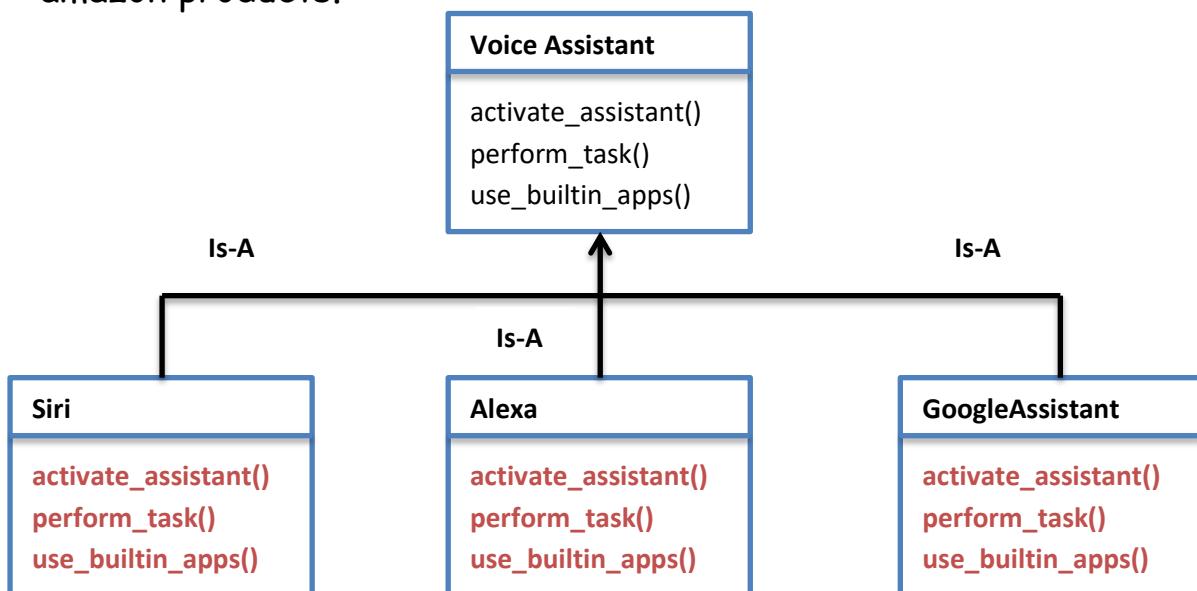
Today's Agenda

- Abstraction
- Rules of abstract class



Abstraction

Let us understand the abstraction by taking an example of voice assistant, we have several voice assistants built-in in our mobile phones like siri in iPhone, google assistant in android phone, alexa in amazon products.



Let us implement the code for the above UML

```
class VoiceAssistant:

    def activate_assistant(self):
        print('VA activated')

    def perform_task(self):
        print('VA is performing the task')

    def use_builtin_apps(self):
        print('VA is using built-in apps')

class Siri(VoiceAssistant):

    def activate_assistant(self):
        print('Hey Siri, activates Siri')

    def perform_task(self):
        print('Siri is performing task using apple servers')

    def use_builtin_apps(self):
        print('Siri uses the builtin apps of ios')

class Alexa(VoiceAssistant):

    def activate_assistant(self):
        print('Alexa, activates Alexa')

    def perform_task(self):
        print('Alexa is performing task using amazon servers')

    def use_builtin_apps(self):
        print('Alexa uses the builtin apps of fire-os')

class GoogleAssistant(VoiceAssistant):

    def activate_assistant(self):
        print('Ok Google, activates GA')

    def perform_task(self):
        print('GA is performing task using google servers')

    def use_builtin_apps(self):
        print('Google uses the builtin apps of android-os')

def main():
    s=Siri()
    a=Alexa()
    ga=GoogleAssistant()

    s.activate_assistant()
    s.perform_task()
    s.use_builtin_apps()

    a.activate_assistant()
    a.perform_task()
    a.use_builtin_apps()

    ga.activate_assistant()
    ga.perform_task()
    ga.use_builtin_apps()

main()
```



Output:

```
In [1]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
Hey Siri, activates Siri
Siri is performing task using apple servers
Siri uses the builtin apps of ios
Alexa, activates Alexa
Alexa is performing task using amazon servers
Alexa uses the builtin apps of fire-os
Ok Google, activates GA
GA is performing task using google servers
Google uses the builtin apps of android-os
```

But in the above code we have just implemented inheritance. We could also implement polymorphism as below

```
class VoiceAssistant:

    def activate_assistant(self):
        print('VA activated')

    def perform_task(self):
        print('VA is performing the task')

    def use_builtin_apps(self):
        print('VA is using built-in apps')

class Siri(VoiceAssistant):

    def activate_assistant(self):
        print('Hey Siri, activates Siri')

    def perform_task(self):
        print('Siri is performing task using apple servers')

    def use_builtin_apps(self):
        print('Siri uses the builtin apps of ios')

class Alexa(VoiceAssistant):

    def activate_assistant(self):
        print('Alexa, activates Alexa')

    def perform_task(self):
        print('Alexa is performing task using amazon servers')

    def use_builtin_apps(self):
        print('Alexa uses the builtin apps of fire-os')
```

```

class GoogleAssistant(VoiceAssistant):

    def activate_assistant(self):
        print('Ok Google, activates GA')

    def perform_task(self):
        print('GA is performing task using google servers')

    def use_builtin_apps(self):
        print('Google uses the builtin apps of android-os')

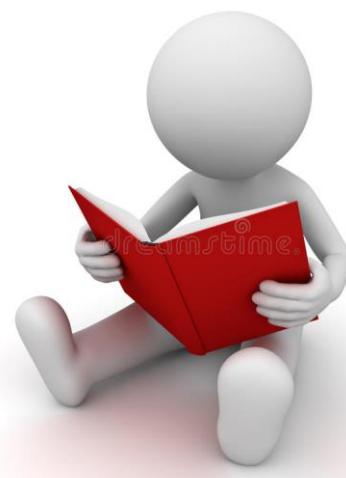
    def use_assistant(ref):
        ref.activate_assistant()
        ref.perform_task()
        ref.use_builtin_apps()

def main():
    s=Siri()
    a=Alexa()
    ga=GoogleAssistant()

    use_assistant(s)
    use_assistant(a)
    use_assistant(ga)

main()

```



Output:

```

In [3]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
Hey Siri, activates Siri
Siri is performing task using apple servers
Siri uses the builtin apps of ios
Alexa, activates Alexa
Alexa is performing task using amazon servers
Alexa uses the builtin apps of fire-os
Ok Google, activates GA
GA is performing task using google servers
Google uses the builtin apps of android-os

```

We see in above code, all the inherited or the child classes have modified the inherited method as required. So can we remove the parent class methods and still get the output as expected? And if we

remove methods from parent class, we can change the names of methods in derived classes. Let us see if we can get the same output

```
class VoiceAssistant:  
    pass  
  
class Siri(VoiceAssistant):  
  
    def start_assistant(self):  
        print('Hey Siri, activates Siri')  
  
    def perform_t(self):  
        print('Siri is performing task using apple servers')  
  
    def builtin_apps(self):  
        print('Siri uses the builtin apps of ios')  
  
class Alexa(VoiceAssistant):  
  
    def initiate_assistant(self):  
        print('Alexa, activates Alexa')  
  
    def do_task(self):  
        print('Alexa is performing task using amazon servers')  
  
    def u_b_a(self):  
        print('Alexa uses the builtin apps of fire-os')  
  
class GoogleAssistant(VoiceAssistant):  
  
    def activate_assistant(self):  
        print('Ok Google, activates GA')  
  
    def perform_task(self):  
        print('GA is performing task using google servers')  
  
    def use_builtin_apps(self):  
        print('Google uses the builtin apps of android-os')  
  
    def use_assistant(ref):  
        ref.activate_assistant()  
        ref.perform_task()  
        ref.use_builtin_apps()  
  
def main():  
    s=Siri()  
    a=Alexa()  
    ga=GoogleAssistant()  
  
    use_assistant(s)  
    use_assistant(a)  
    use_assistant(ga)  
  
main()
```



Output:

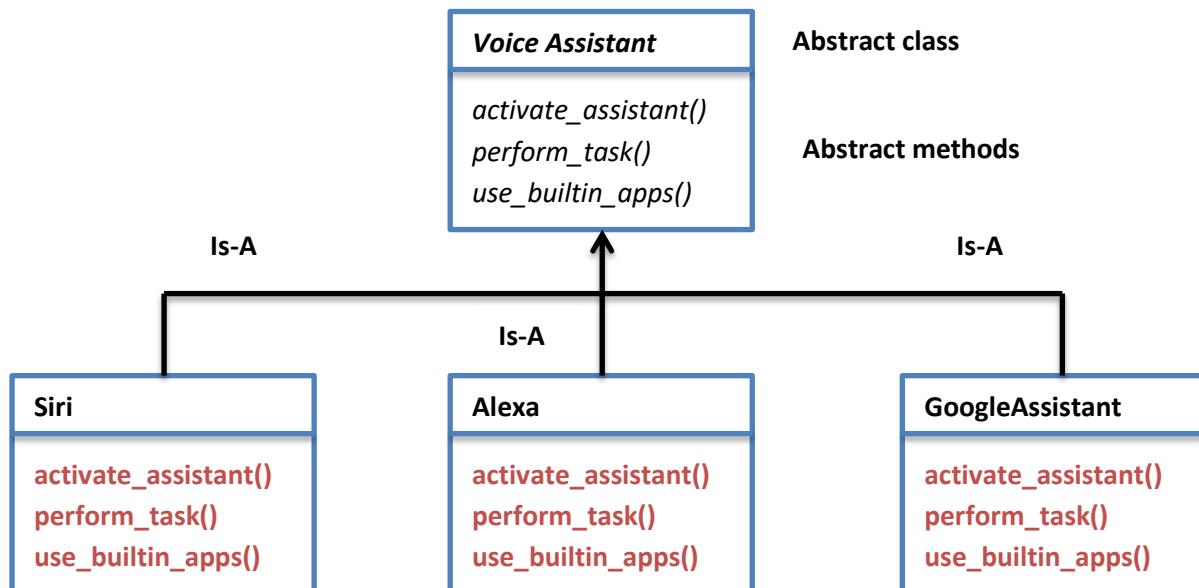
```
File "C:/Users/rooman/OneDrive/Desktop/python/test.py",
line 47, in main
    use_assistant(s)

File "C:/Users/rooman/OneDrive/Desktop/python/test.py",
line 38, in use_assistant
    ref.activate_assistant()

AttributeError: 'Siri' object has no attribute
'activate_assistant'
```

We are getting error because ref was previously pointing to the overridden methods, but now those methods have become specialised methods. So let us undo the changes done and as body is the only thing that is not being used by any of the derived or inherited or child class, let us remove the body of parent class methods.

The methods which contains only the name is called as abstract methods and a class which contains abstract methods are called as abstract class.



```

from abc import ABC,abstractmethod
class VoiceAssistant(ABC): #abstract base class

    @abstractmethod
    def activate_assistant(self):
        pass

    @abstractmethod
    def perform_task(self):
        pass
    @abstractmethod
    def use_builtin_apps(self):
        pass

class Siri(VoiceAssistant):

    def activate_assistant(self):
        print('Hey Siri, activates Siri')

    def perform_task(self):
        print('Siri is performing task using apple servers')

    def use_builtin_apps(self):
        print('Siri uses the builtin apps of ios')

class Alexa(VoiceAssistant):

    def activate_assistant(self):
        print('Alexa, activates Alexa')

    def perform_task(self):
        print('Alexa is performing task using amazon servers')

    def use_builtin_apps(self):
        print('Alexa uses the builtin apps of fire-os')

class GoogleAssistant(VoiceAssistant):

    def activate_assistant(self):
        print('Ok Google, activates GA')

    def perform_task(self):
        print('GA is performing task using google servers')

    def use_builtin_apps(self):
        print('Google uses the builtin apps of android-os')

def use_assistant(ref):
    ref.activate_assistant()
    ref.perform_task()
    ref.use_builtin_apps()

def main():
    s=Siri()
    a=Alexa()
    ga=GoogleAssistant()

    use_assistant(s)
    use_assistant(a)
    use_assistant(ga)

main()

```



Output:

```
In [5]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
Hey Siri, activates Siri
Siri is performing task using apple servers
Siri uses the builtin apps of ios
Alexa, activates Alexa
Alexa is performing task using amazon servers
Alexa uses the builtin apps of fire-os
Ok Google, activates GA
GA is performing task using google servers
Google uses the builtin apps of android-os
```

Abstract methods are used when you are not sure about the body of method of the derived classes.

Rules of abstract classes

- ❖ An abstract class can contain both concrete as well as abstract methods.

```
from abc import ABC,abstractmethod
class VoiceAssistant(ABC): #abstract base class

    @abstractmethod
    def activate_assistant(self):
        pass

    @abstractmethod
    def perform_task(self):
        pass
    @abstractmethod
    def use_builtin_apps(self):
        pass

    def fun(self): #concrete method
        print('Inside fun()')
```



Output:

```
In [6]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
```

In [7]:

Certainly we haven't called any method so we won't get anything on output screen. The point is to see if it works or throws error.

- ❖ If a class is abstract class, its objects cannot be created.

```
from abc import ABC,abstractmethod
class VoiceAssistant(ABC): #abstract base class

    @abstractmethod
    def activate_assistant(self):
        pass

    @abstractmethod
    def perform_task(self):
        pass
    @abstractmethod
    def use_builtin_apps(self):
        pass

    def fun(self): #concrete method
        print('Inside fun()')

va=VoiceAssistant() #object creation
```

Output:

```
File "C:/Users/rooman/OneDrive/Desktop/python/test.py",
line 18, in <module>
    va=VoiceAssistant()
```

```
TypeError: Can't instantiate abstract class VoiceAssistant
with abstract methods activate_assistant, perform_task,
use_builtin_apps
```

This is because, if the behaviour of object is unknown what is the use of creating such object.

- ❖ A child class if inherited from a parent class which is abstract can only create an object if it overrides all the methods inherited from parent class.



```

from abc import ABC,abstractmethod
class VoiceAssistant(ABC): #abstract base class

    @abstractmethod
    def activate_assistant(self):
        pass

    @abstractmethod
    def perform_task(self):
        pass
    @abstractmethod
    def use_builtin_apps(self):
        pass

    def fun(self): #concrete method
        print('Inside fun()')

class Siri(VoiceAssistant):

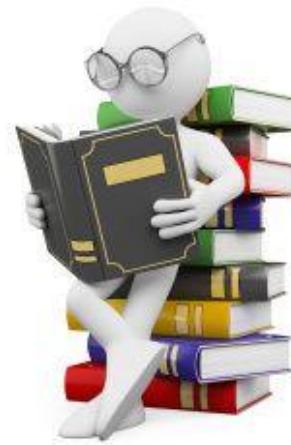
    def activate_assistant(self):
        print('Hey Siri')

    def perform_task(self):
        print('Performing task')

    def use_builtin_apps(self):
        print('Using builtin apps')

s=Siri() #object of child class
s.activate_assistant()
s.fun()
s.perform_task()
s.use_builtin_apps()

```



Output:

```

In [8]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
Hey Siri
Inside fun()
Performing task
Using builtin apps

```

Python fundamentals

day 60

Today's agenda

- Abstraction contd
- Problems on abstraction



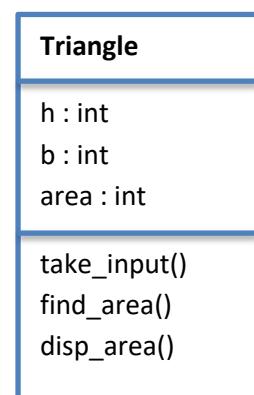
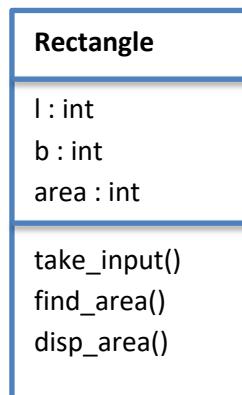
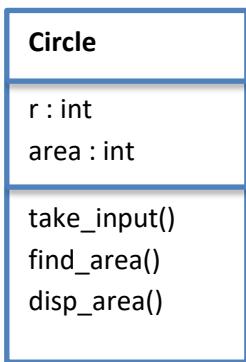
Abstraction contd

Let us take an example of shapes and try to understand when a method must be abstract, incomplete or a concrete method

$$\text{Area} = \pi r^2$$

$$\text{Area} = l * b$$

$$\text{Area} = (h * b) / 2$$



```

from math import pi
class Circle:
    def __init__(self):
        self.r=0
        self.area=0

    def take_input(self):
        self.r=int(input("Enter the radius: \n"))

    def find_area(self):
        self.area=pi*self.r**2

    def disp_area(self):
        print(f'circle area = {self.area}')

class Rectangle:
    def __init__(self):
        self.l=0
        self.b=0
        self.area=0

    def take_input(self):
        self.l=int(input("Enter the length: \n"))
        self.b=int(input("Enter the breadth: \n"))

    def find_area(self):
        self.area=self.l*self.b

    def disp_area(self):
        print(f'Rectangle area = {self.area}')

class Triangle:
    def __init__(self):
        self.h=0
        self.b=0
        self.area=0

    def take_input(self):
        self.h=int(input("Enter the height: \n"))
        self.b=int(input("Enter the base: \n"))

    def find_area(self):
        self.area=(self.h*self.b)/2

    def disp_area(self):
        print(f'Triangle area = {self.area}')

def main():
    c=Circle()
    r=Rectangle()
    t=Triangle()

    c.take_input()
    c.find_area()
    c.disp_area()

    r.take_input()
    r.find_area()
    r.disp_area()

    t.take_input()
    t.find_area()
    t.disp_area()

main()

```



Output:

```
In [1]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')

Enter the radius:
7
circle area = 153.93804002589985

Enter the length:
10

Enter the breadth:
20
Rectangle area = 200

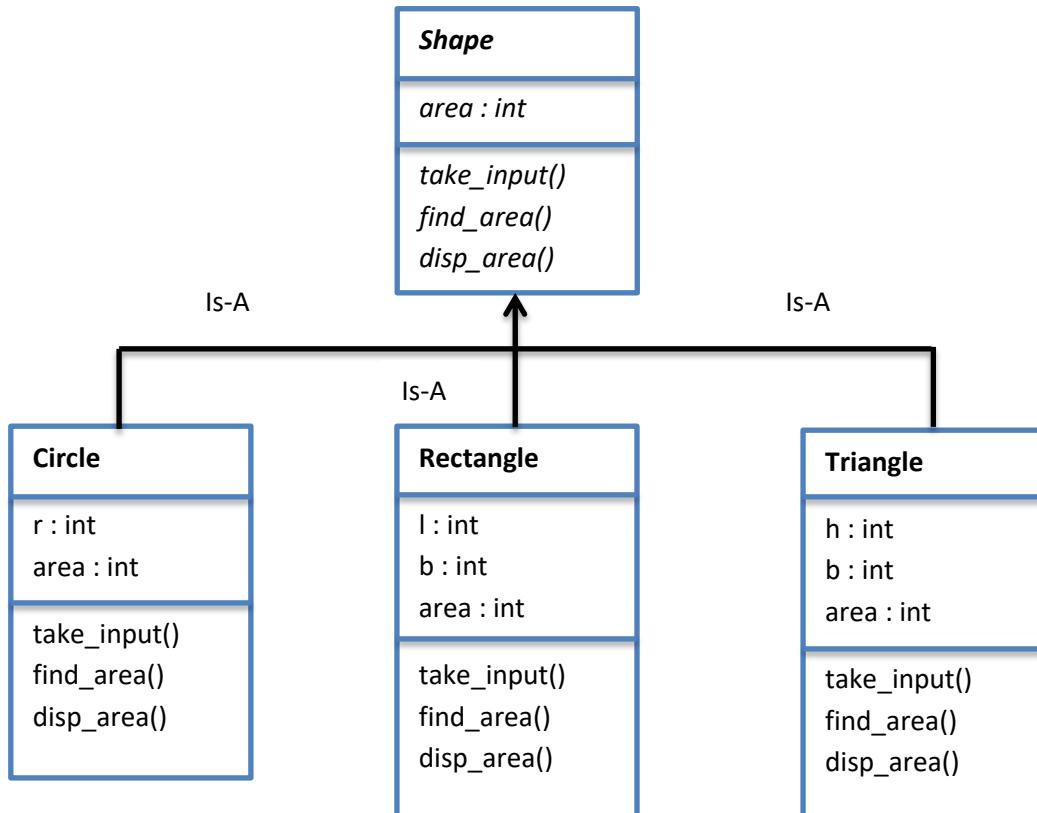
Enter the height:
5

Enter the base:
10
Triangle area = 25.0
```



But we see the above code is not object oriented, and does not exhibit any features of object orientation like inheritance, polymorphism and abstraction.

Let us try to code using these features



```

from math import pi
from abc import ABC,abstractmethod

class Shape(ABC):

    def __init__(self):
        self.area=0

    @abstractmethod
    def take_input(self):
        pass

    @abstractmethod
    def find_area(self):
        pass

    @abstractmethod
    def disp_area(self):
        pass

class Circle(Shape):
    def __init__(self):
        self.r=0

    def take_input(self):
        self.r=int(input("Enter the radius: \n"))

    def find_area(self):
        self.area=pi*self.r**2

    def disp_area(self):
        print(f'circle area = {self.area}')


class Rectangle(Shape):
    def __init__(self):
        self.l=0
        self.b=0
        super().__init__()

    def take_input(self):
        self.l=int(input("Enter the length: \n"))
        self.b=int(input("Enter the breadth: \n"))

    def find_area(self):
        self.area=self.l*self.b

    def disp_area(self):
        print(f'Rectangle area = {self.area}')

```



```

class Triangle(Shape):
    def __init__(self):
        self.h=0
        self.b=0
        super().__init__()

    def take_input(self):
        self.h=int(input("Enter the height: \n"))
        self.b=int(input("Enter the base: \n"))

    def find_area(self):
        self.area=(self.h*self.b)/2

    def disp_area(self):
        print(f'Triangle area = {self.area}')

    def geometric_shapes(self):
        self.take_input()
        self.find_area()
        self.disp_area()

def main():
    c=Circle()
    r=Rectangle()
    t=Triangle()

    geometric_shapes(c)
    geometric_shapes(r)
    geometric_shapes(t)

main()

```

Output:

```

In [3]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')

Enter the radius:
7
circle area = 153.93804002589985

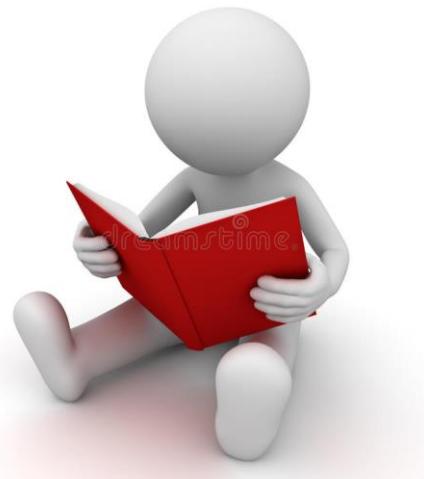
Enter the length:
10

Enter the breadth:
20
Rectangle area = 200

Enter the height:
5

Enter the base:
10
Triangle area = 25.0

```



Problems on abstraction

Example 1: Let's try a larger application like fund transfer before moving on to our application. So, in fund transfer there are three types NEFT/IMPS/PTGS.

We can create an abstract class FundTransfer and extend it in the child classes. Create an abstract method transfer and implement in all the child classes.

Create an abstract class FundTransfer with following attributes, accountNumber:

int, balance : float and following methods,

validate(amount) : to check if the accountNumber is 10 digits, transfer the amount in non-negative and less than balance, and return true, if not false

transfer (amount) : abstract method with no definition

Create a class NEFTTransfer which extends FundTransfer and implements transfer method,

transfer(amount) : Check if transfer amount + 5% of transfer amount is less than balance, then subtract transfer amount and 5% service charge from balance and return true, if not return false

Create a class IMPSTransfer which extends FundTransfer and implements transfer method,

transfer(amount) : Check if the transfer amount + 2% of transfer amount is less than balance, then subtract transfer amount and 2% service charge from balance and return true, if not return false

Create a class RTGSTransfer which extends FundTransfer and implements Transfer method,

transfer(amount) : Check if transfer amount is greater than Rs.10000, then subtract the transfer amount from balance and return true, if not return false

Add appropriate getters/setters, constructors with super() to create objects

Note: Print "Account number or transfer amount seems to be wrong" if validate function returns false.

Print "Transfer could not be made" if transfer function returns false.

Sample Input/Output:

Enter your account number: 1234567890

Enter the balance of the amount: 10000

Enter the type of transfer to be made:

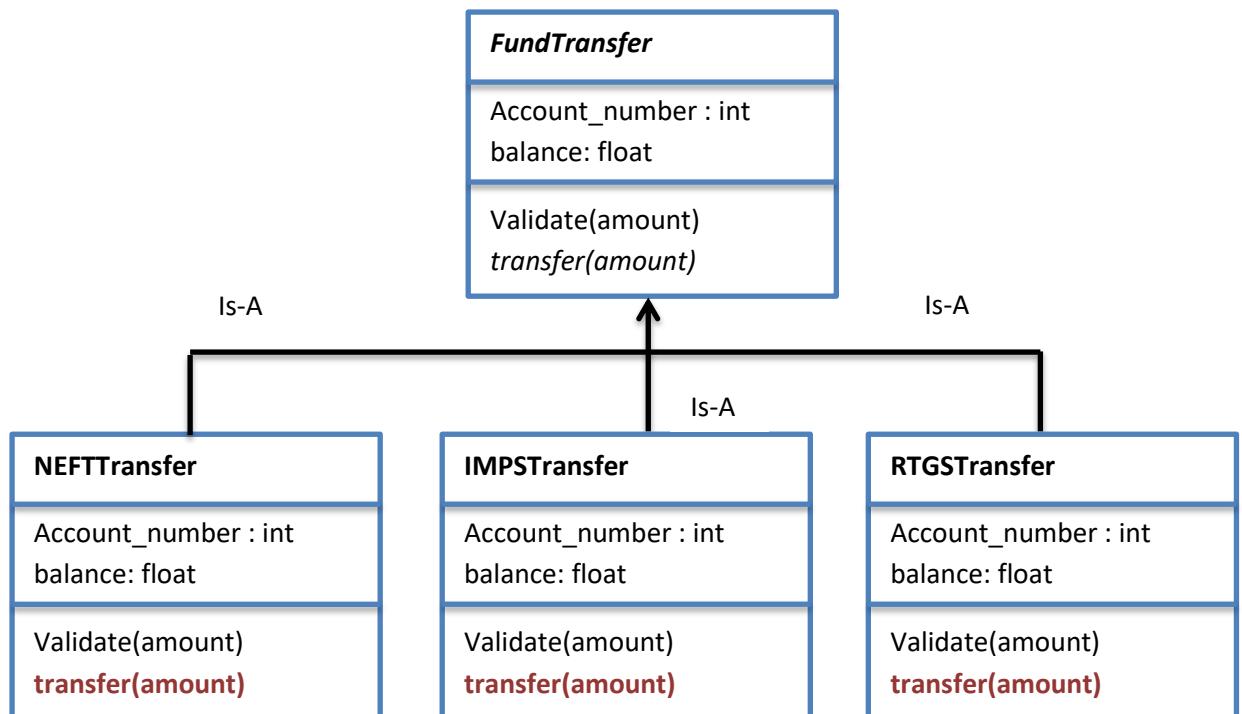
1. NEFT
2. IMPS
3. RTGS

1

Enter the amount to be transferred : 2000

Transfer occurred successfully remaining balance is 7900.0





```

from abc import ABC, abstractmethod

class FundTransfer(ABC):

    def __init__(self, account_number, balance):
        self.__account_number=account_number
        self.__balance=balance

    @property
    def account_number(self):
        return self.__account_number

    @account_number.setter
    def account_number(self, account_number):
        if len(str(account_number))==10:
            self.__account_number=account_number

    @property
    def balance(self):
        return self.__balance

    @balance.setter
    def balance(self, balance):
        if balance>0:
            self.__balance=balance

    def validate(self, amount):
        return len(str(self.account_number))==10 and \
               amount<self.__balance and amount>0

    @abstractmethod
    def transfer(self, amount):
        pass

```



dreamstime.

```

class NEFTTransfer(FundTransfer):
    def __init__(self,account_number,balance):
        super().__init__(account_number,balance)

    def transfer(self,amount):
        sc=amount*0.05
        if (amount+sc)<self.balance:
            self.balance=self.balance-(amount+sc)
            return True
        return False

class IMPSTransfer(FundTransfer):
    def __init__(self,account_number,balance):
        super().__init__(account_number,balance)

    def transfer(self,amount):
        sc=amount*0.02
        if (amount+sc)<self.balance:
            self.balance=self.balance-(amount+sc)
            return True
        return False

class RTGSTransfer(FundTransfer):
    def __init__(self,account_number,balance):
        super().__init__(account_number,balance)

    def transfer(self,amount):
        if amount<self.balance and amount>=10000:
            self.balance=self.balance-amount
            return True
        return False

def main():
    an=int(input("Enter your account number: "))
    bal=int(input("Enter your account balance: "))

    print("Enter your choice")
    print("1-NEFT\n2-IMPS\n3-RTGS\n")
    choice=int(input())

    if choice==1:
        ref=NEFTTransfer(an,bal)
    elif choice==2:
        ref=IMPSTransfer(an,bal)
    elif choice==3:
        ref=RTGSTransfer(an,bal)
    else:
        print('INVALID CHOICE')

    amt=int(input('Enter the amount to be transferred:'))

    if ref.validate(amt):
        if ref.transfer(amt):
            print('Transfer occurred successfully')
            print(f'Remaining balance is {ref.balance}')
        else:
            print('Transfer could not be made')
    else:
        print('Account number or transfer amount seems to be wrong')

if __name__=='__main__':
    main()

```



Output:

```
In [2]: runfile('C:/Users/rooman/Downloads/test.py',  
      wdir='C:/Users/rooman/Downloads')
```

Enter your account number: 1010235689

Enter your account balance: 1000000

Enter your choice

1-NEFT

2-IMPS

3-RTGS

1

Enter the amount to be transferred:8000

Transfer occurred successfully

Remaining balance is 991600.0

```
In [3]: runfile('C:/Users/rooman/Downloads/test.py',  
      wdir='C:/Users/rooman/Downloads')
```

Enter your account number: 1010235689

Enter your account balance: 100000

Enter your choice

1-NEFT

2-IMPS

3-RTGS

3

Enter the amount to be transferred:8000

Transfer could not be made

As RTGS expects minimum of Rs.10000 transfer could not be made.



Python Fundamentals

day 58

Today's Agenda

- Sorting
- Sorting algorithms



Sorting

Sorting is a process of arranging items systematically, be it in ascending order or descending order. In python we can sort built-in objects as well as user defined objects.

Let us understand with the basic example first

```
lst=[20,50,40,30,10,60]
print(lst)
lst.sort()
print(lst)
```

Output:

```
In [1]: runfile('C:/Users/rooman/OneDrive/Desktop/python/
test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
[20, 50, 40, 30, 10, 60]
[10, 20, 30, 40, 50, 60]
```

By default the elements in the list get sorted in ascending order.
What to do to get it in descending order let us see below

```
lst=[20,50,40,30,10,60]
print(lst)
lst.sort(reverse=True)
print(lst)
```

Output:

```
In [2]: runfile('C:/Users/rooman/OneDrive/Desktop/python/
test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
[20, 50, 40, 30, 10, 60]
[60, 50, 40, 30, 20, 10]
```

Next let us try with strings

```
lst=['python','java','c','abc','ruby']
print(lst)
lst.sort()
print(lst)
```

Output:

```
In [3]: runfile('C:/Users/rooman/OneDrive/Desktop/python/
test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
['python', 'java', 'c', 'abc', 'ruby']
['abc', 'c', 'java', 'python', 'ruby']
```

The string values are sorted in alphabetical order. And just like numbers we can have the reverse order of strings as well

```
lst=['python','java','c','abc','ruby']
print(lst)
lst.sort(reverse=True)
print(lst)
```

Output:

```
In [4]: runfile('C:/Users/rooman/OneDrive/Desktop/python/
test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
['python', 'java', 'c', 'abc', 'ruby']
['ruby', 'python', 'java', 'c', 'abc']
```

So far we have sorted homogeneous data, what if we provide heterogeneous data? Combination of multiple datatypes. Let us see

```
lst=['python',30,10,'java','c',20,'abc',40,'ruby']
print(lst)
lst.sort(reverse=True)
print(lst)
```



Output:

```
File "C:/Users/rooman/OneDrive/Desktop/python/test.py",
line 3, in <module>
    lst.sort(reverse=True)

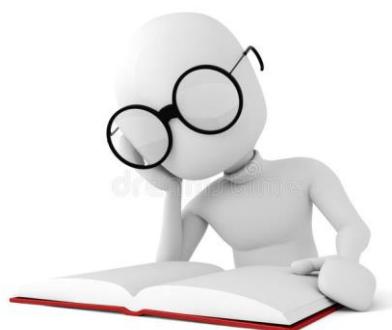
TypeError: '<' not supported between instances of 'int'
and 'str'
```

So as we can see heterogeneous sorting cannot happen, because we cannot measure two different set of data.

Sorting Algorithms

There are different algorithms (step by step procedure) used for sorting, below are few algorithms mentioned

- ❖ Bubble sort
- ❖ Mere sort
- ❖ Insertion sort
- ❖ Quick sort
- ❖ Selection sort
- ❖ Heap sort
- ❖ Radix sort
- ❖ Bucket sort



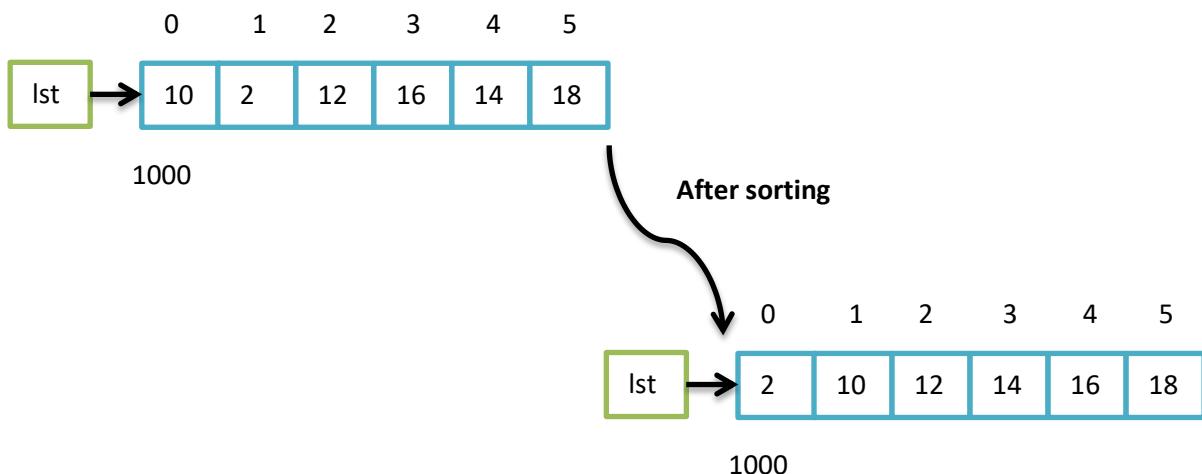
Amongst all let us select selection sort and know about it.

Selection sort:

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

- 1)The subarray which is already sorted.
- 2) Remaining subarray which is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.



Boundary condition

```
i : range(0,len(lst)-1)  
j : range(i+1, len(lst))
```

Condition

```
if lst[j]<lst[i]:  
    lst[i],lst[j]=lst[j],lst[i]
```

Let us try to implement the above code:



```

lst=[18,12,2,16,10,14]
print(lst)

for i in range(0,len(lst)-1):
    for j in range(i+1, len(lst)):
        if lst[j]<lst[i]:
            lst[i],lst[j]=lst[j],lst[i]

print(lst)

```

Output:

```

In [6]: runfile('C:/Users/rooman/OneDrive/Desktop/python/
test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
[18, 12, 2, 16, 10, 14]
[2, 10, 12, 14, 16, 18]

```



Next let us see the selection sort for user defined objects. For that first we have to create the user defined objects and then shall proceed with sorting

```

class Footballer:

    def __init__(self,name,goals,asist):
        self.name=name
        self.goals=goals
        self.asist=asist

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

def main():
    f1=Footballer('Messi',650,359)
    f2=Footballer('Cristiano',750,250)
    f1.display()
    f2.display()

main()

```

Output:

```

In [7]: runfile('C:/Users/rooman/OneDrive/Desktop/python/
test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
{'name': 'Messi', 'goals': 650, 'asist': 359}
{'name': 'Cristiano', 'goals': 750, 'asist': 250}

```



The user defined objects are ready, but can we just sort them by using any operators we used for built-in objects? Certainly not! So what to do?

```
class Footballer:

    def __init__(self, name, goals, assist):
        self.name = name
        self.goals = goals
        self.assist = assist

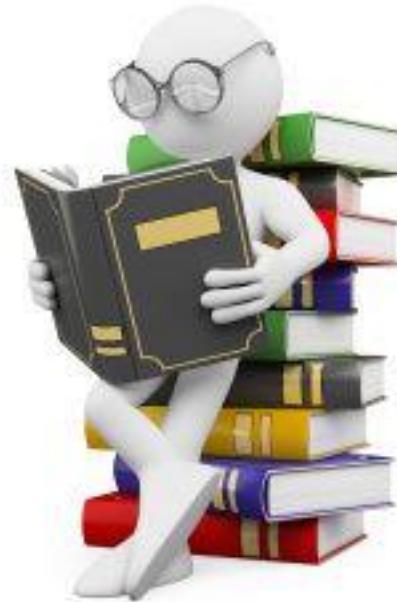
    def display(self):
        print(self.__dict__)

    def __lt__(self, other):
        if self.goals < other.goals:
            return True
        else:
            return False

    def __gt__(self, other):
        if self.goals > other.goals:
            return True
        else:
            return False

def main():
    f1 = Footballer('Messi', 650, 359)
    f2 = Footballer('Cristiano', 750, 250)
    f1.display()
    f2.display()
    print(f1 < f2) #f1.__lt__f2
    print(f1 > f2) #f1.__gt__f2

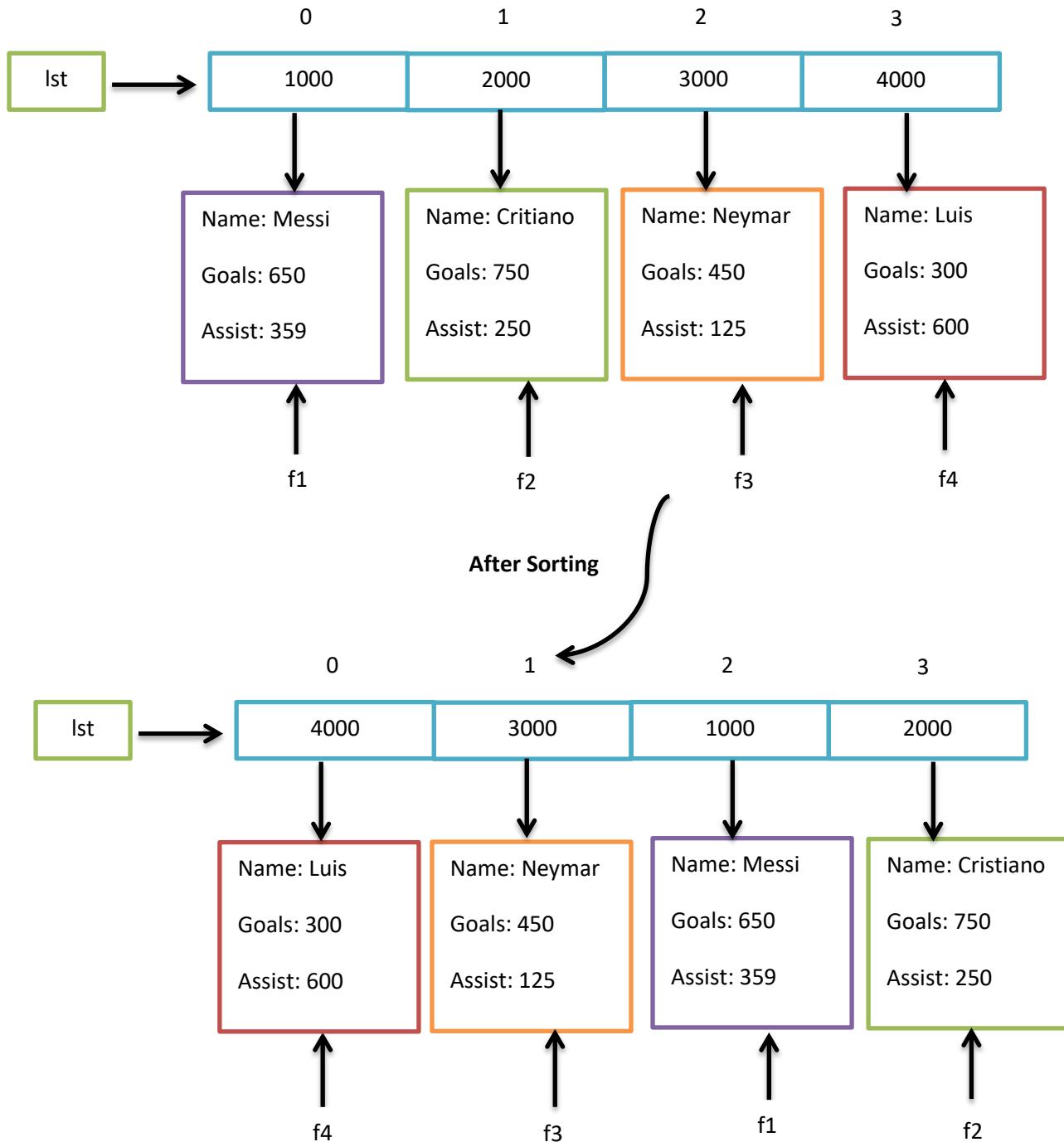
main()
```



Output:

```
In [8]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
{'name': 'Messi', 'goals': 650, 'assist': 359}
{'name': 'Cristiano', 'goals': 750, 'assist': 250}
True
False
```

Now that we know how to compare user defined objects, let us see how to sort them



```

class Footballer:

    def __init__(self, name, goals, assist):
        self.name = name
        self.goals = goals
        self.assist = assist

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

    def __lt__(self, other):
        if self.goals < other.goals:
            return True
        else:
            return False

    def __gt__(self, other):
        if self.goals > other.goals:
            return True
        else:
            return False

    def __str__(self):
        return f'{self.name} {self.goals} {self.assist}'

def sort_footballer(lst):
    for i in range(0, len(lst) - 1):
        for j in range(i + 1, len(lst)):
            if lst[j] < lst[i]:
                lst[i], lst[j] = lst[j], lst[i]

def main():
    f1 = Footballer('Messi', 650, 359)
    f2 = Footballer('Cristiano', 750, 250)
    f3 = Footballer('Neymar', 450, 125)
    f4 = Footballer('Luis', 300, 600)
    lst = [f1, f2, f3, f4]
    sort_footballer(lst)

    for i in lst:
        print(i)

main()

```

Output:

```

In [13]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
Luis 300 600
Neymar 450 125
Messi 650 359
Cristiano 750 250

```

Great! But is this the optimized code? Definitely not

```
class Footballer:

    def __init__(self, name, goals, assist):
        self.name = name
        self.goals = goals
        self.assist = assist

    def __lt__(self, other):
        return self.goals < other.goals

    def __gt__(self, other):
        return self.goals > other.goals

    def __str__(self):
        return f'{self.name} {self.goals} {self.assist}'

def main():
    f1 = Footballer('Messi', 650, 359)
    f2 = Footballer('Cristiano', 750, 250)
    f3 = Footballer('Neymar', 450, 125)
    f4 = Footballer('Luis', 300, 600)
    lst = [f1, f2, f3, f4]
    lst.sort()

    for i in lst:
        print(i)

main()
```



Output:

```
In [14]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
Luis 300 600
Neymar 450 125
Messi 650 359
Cristiano 750 250
```

Python Fundamentals

day 61

Today's Agenda

- Exception handling



Exception handling

In computing and computer programming, exception handling is the process of responding to the occurrence of exceptions - anomalous or exceptional conditions requiring special processing - during the execution of a program.

To understand in a better way let us take non-technical example first

Consider the following sentence:

A apple a day keep a doctor away.

Clearly we can see some grammatical mistakes in the above sentence. The correct form of sentence is "An apple a day keeps the doctor away". This follows all the set of rules which English language follows. Similar to these grammatical mistakes in programming language one can commit some syntactical or logical mistakes.

In python we have set of tokens (identifiers, operators, delimiters, keywords). All commands in python are combination of these tokens.

Tokens
Identifiers: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z a b c d e f g h i j k l m n o p q r s t u v w x y z 0 1 2 3 4 5 6 7 8 9
Operators: + - * / % ** // << >> & ^ ~ < <= > >= <> != ==
Delimiters: () {} [] , : . = ; += -= *= /= //=%= &= = ^= >=>= <<= **=
Keywords: and del for is raise assert elif from lambda return break else global not try class except if or while continue exec import pass with def finally in print yield

Consider the following statement

```
for i in range(6):
```

```
    print(i)
```

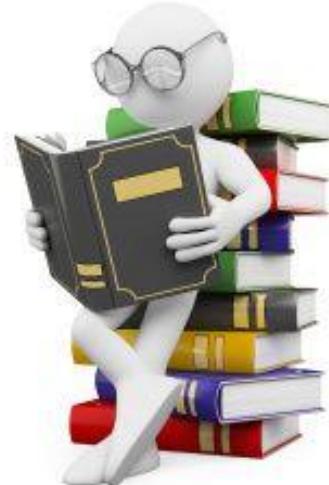
Above statement is correct according to the rules of python. Let us take another statement

```
fun(x):
```

```
    print(x)
```

The above statement is invalid as it is not following the syntax of a normal function. The above statement will cause syntax error which can be fixed by following the rule and adding def keyword before the fun(x).

But this is not exception. The mistakes occurred due to syntax are called syntactical errors. But if the mistake is with the logic then it is called as an exception. These exceptions occur during the execution of the program and once an exception is encountered



program gets abruptly terminated. To make sure it doesn't get abruptly terminated we have to handle these exceptions.

For eg:

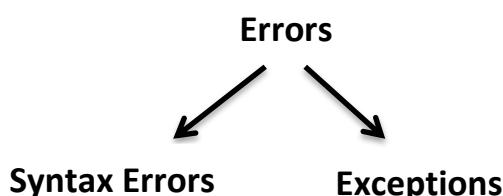
```
a=10+0  
b=10-0  
c=10*0  
d=10/0  
print(a,b,c,d)
```



Output:

```
In [1]: runfile('C:/Users/rooman/OneDrive/Desktop/python/  
test1.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')  
Traceback (most recent call last):  
  
  File "<ipython-input-1-33651a4fccdf>", line 1, in  
<module>  
    runfile('C:/Users/rooman/OneDrive/Desktop/python/  
test1.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')  
  
  File "C:\Users\rooman\Anaconda3\lib\site-packages  
\spyder_kernels\customize\spydercustomize.py", line 827,  
in runfile  
    execfile(filename, namespace)  
  
  File "C:\Users\rooman\Anaconda3\lib\site-packages  
\spyder_kernels\customize\spydercustomize.py", line 110,  
in execfile  
    exec(compile(f.read(), filename, 'exec'), namespace)  
  
  File "C:/Users/rooman/OneDrive/Desktop/python/test1.py",  
line 4, in <module>  
    d=10/0  
  
ZeroDivisionError: division by zero
```

Above code is perfectly fine with the syntax. But logic is not correct. Hence we get a ZeroDivisionError which is one of the exceptions.



Syntax Errors	Exceptions
<ul style="list-style-type: none"> • Syntax mistakes 	<ul style="list-style-type: none"> • Logical mistakes
<ul style="list-style-type: none"> • Is easily identified by the interpreter 	<ul style="list-style-type: none"> • Is not identified by the interpreter
<ul style="list-style-type: none"> • Occurs before runtime 	<ul style="list-style-type: none"> • Occurs during runtime

Let us start by considering an example of banking

```
print('Secure connection has been established to bank server')

p=int(input('Enter your principal amount'))
t=int(input('Enter the duration'))
r=10
si=(p*t*r)/100
print('Simple interest =',si)
print('Secure connection has been closed to the bank server')
```

Output:

```
In [2]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test1.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
Secure connection has been established to bank server

Enter your principal amount1000000
Enter the duration2
Simple interest = 200000.0
Secure connection has been closed to the bank server
```

The above code is working fine, as no logical errors occur during execution. But imagine if the input instead of giving in number if it was entered in string. What would happen? Let us see

```
File "C:\Users\rooman\Anaconda3\lib\site-packages
\spyder_kernels\customize\spydercustomize.py", line 110,
in execfile
    exec(compile(f.read(), filename, 'exec'), namespace)

File "C:/Users/rooman/OneDrive/Desktop/python/test1.py",
line 3, in <module>
    p=int(input('Enter your principal amount'))

ValueError: invalid literal for int() with base 10: 'ten
lakhs'          Activate Windows
```

It has definitely abruptly terminated and the lines below have not executed. More importantly the connection is not been closed. This may result in mischievous problem.

Let us see how can we handle these exceptions and can achieve complete execution of code

- Look for the statement which might arise in logical error. Especially if the input is taken from the users.
- Put those set of instructions with a special block namely "try" and "except" as shown below

```
print('Secure connection has been established to bank server')

try:
    p=int(input('Enter your principal amount'))
    t=int(input('Enter the duration'))
    r=10
    si=(p*t*r)/100
    print('Simple interest =',si)
except:
    print('Please provide the numerical value')

print('Secure connection has been closed to the bank server')
```

Output:

```
In [4]: runfile('C:/Users/rooman/OneDrive/Desktop/python/
test1.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
Secure connection has been established to bank server

Enter your principal amountten lakhs
Please provide the numerical value
Secure connection has been closed to the bank server
```

Even though we did not get the expected output, the code executed completely and the connection was closed. Along with it the error message made sense as to why we did not get the expected output.

Note: except block executes only when exception occurs in the try block and an exception object is generated. As shown in below case

```
In [5]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test1.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
Secure connection has been established to bank server
```

```
Enter your principal amount1000000
```

```
Enter the duration2
```

```
Simple interest = 200000.0
```

```
Secure connection has been closed to the bank server
```

There is another block which can be added namely "else" block, under which those statements have to be present which must execute if no exceptions occur in "try" block. As shown below

```
print('Secure connection has been established to bank server')

try:
    p=int(input('Enter your principal amount'))
    t=int(input('Enter the duration'))
    r=10
except:
    print('Please provide the numerical value')
else:
    si=(p*t*r)/100
    print('Simple interest =',si)

print('Secure connection has been closed to the bank server')
```

Output:

```
In [6]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test1.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
Secure connection has been established to bank server
```

```
Enter your principal amount1000000
```

```
Enter the duration2
```

```
Simple interest = 200000.0
```

```
Secure connection has been closed to the bank server
```

Python fundamentals

day 62

Today's agenda

- Exception handling contd



Exception handling contd

Let us take another example

```
print('Execution started normally')
lst=[10,20,0,40,50]
d={1:'c',2:'java',3:'python',4:'c++'}

try:
    r=int(input('Enter the rank of language:\n'))
    print(d[r])
    num=int(input('Enter the index of numerator:'))
    den=int(input('Enter the index of denominator:'))
    print(lst[num]/lst[den])
except:
    print('Hey there was an issue!')

print('Execution terminated normally')
```



Output:

```
In [1]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test1.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
Execution started normally
```

```
Enter the rank of language:
3
python
```

```
Enter the index of numerator:3
```

```
Enter the index of denominator:0
4.0
Execution terminated normally
```



This is the ideal case. Let us try with other inputs

```
In [2]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test1.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
Execution started normally
```

```
Enter the rank of language:
5
Hey there was an issue!
Execution terminated normally
```

In the above case we have given the key which is not present in our dictionary. Therefore KeyError exception object is generated and it checks for except block and completes execution. But the message is not very clear. Let us see another case

```
In [3]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test1.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
Execution started normally
```

```
Enter the rank of language:
3
python
```

```
Enter the index of numerator:5
```

```
Enter the index of denominator:3
Hey there was an issue!
Execution terminated normally
```



In the above case we don't have 5th index in list. Therefore while performing division IndexError exception object is generated and it checks for the except block and completes the execution. But here as well we get the same message and it doesn't convey the clear message. Let us check for another case

```
In [4]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test1.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
Execution started normally
```

```
Enter the rank of language:
3
python
```

```
Enter the index of numerator:3
```

```
Enter the index of denominator:2
Hey there was an issue!
Execution terminated normally
```

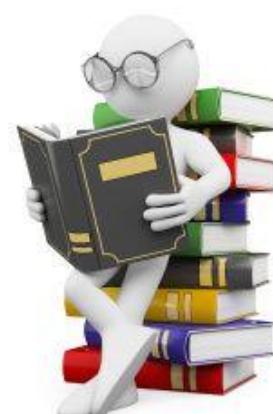
Here we are trying to divide 3rd index by 2nd. 2nd index is 0 which will result in ZeroDivisionError exception which will check for except block and return the message.

But as seen in above cases all exceptions generated the same message which is not conveying the issue properly and hence we need multiple except block for respective exceptions. As shown below

```
print('Execution started normally')
lst=[10,20,0,40,50]
d={1:'c',2:'java',3:'python',4:'c++'}

try:
    r=int(input('Enter the rank of language:\n'))
    print(d[r])
    num=int(input('Enter the index of numerator:'))
    den=int(input('Enter the index of denominator:'))
    print(lst[num]/lst[den])
except KeyError:
    print('Key does not exist')
except IndexError:
    print('Index out of bounds')
except ZeroDivisionError:
    print('Division by zero')

print('Execution terminated normally')
```



Let us try giving the same previous inputs and check the output now

```
In [5]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test1.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
Execution started normally
```

```
Enter the rank of language:
5
Key does not exist
Execution terminated normally
```

Great! Now we know the entered key does not exist. And can correct it by giving proper key. And note that once a particular except block is executed other except blocks do not execute.

```
In [7]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test1.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
Execution started normally
```

```
Enter the rank of language:
3
python
```

```
Enter the index of numerator:5
```

```
Enter the index of denominator:3
Index out of bounds
Execution terminated normally
```

Now we know the issue is with the index number. We can now give appropriate input.

```
In [8]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test1.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
Execution started normally
```

```
Enter the rank of language:
3
python
```

```
Enter the index of numerator:3
```

```
Enter the index of denominator:2
Division by zero
Execution terminated normally
```



And now the issue is with Zero division.

But the above except block will catch only specific exception objects. What if an exception generated cannot be handled by any of the above except blocks? Let us check

```
File "C:\Users\rooman\Anaconda3\lib\site-packages
\spyder_kernels\customize\spydercustomize.py", line 110,
in execfile
    exec(compile(f.read(), filename, 'exec'), namespace)

File "C:/Users/rooman/OneDrive/Desktop/python/test1.py",
line 6, in <module>
    r=int(input('Enter the rank of language:\n'))

ValueError: invalid literal for int() with base 10: 'five'
```

Certainly abrupt termination occurs. To avoid this it is a good habit to add a generic except block which accept all the possible exception objects as shown below

```
print('Execution started normally')
lst=[10,20,0,40,50]
d={1:'c',2:'java',3:'python',4:'c++'}

try:
    r=int(input('Enter the rank of language:\n'))
    print(d[r])
    num=int(input('Enter the index of numerator:'))
    den=int(input('Enter the index of denominator:'))
    print(lst[num]/lst[den])
except KeyError:    #specific handlers
    print('Key does not exist')
except IndexError:
    print('Index out of bounds')
except ZeroDivisionError:
    print('Division by zero')
except:    #generic handler
    print('Hey some issue occurred')

print('Execution terminated normally')
```



```
In [10]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test1.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
Execution started normally

Enter the rank of language:
five
Hey some issue occurred
Execution terminated normally
```

Great! Now we know how to tackle all problems. But now what if you want to place the generic except block on top of all except blocks;

```
print('Execution started normally')
lst=[10,20,0,40,50]
d={1:'c',2:'java',3:'python',4:'c++'}

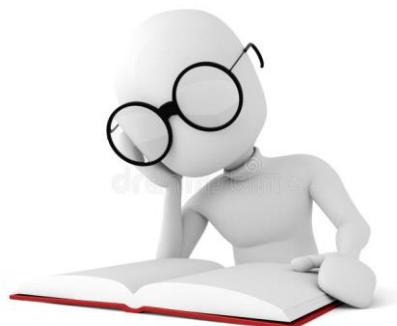
try:
    r=int(input('Enter the rank of language:\n'))
    print(d[r])
    num=int(input('Enter the index of numerator:'))
    den=int(input('Enter the index of denominator:'))
    print(lst[num]/lst[den])
except: #generic handler
    print('Hey some issue occurred')
except KeyError: #specific handlers
    print('Key does not exist')
except IndexError:
    print('Index out of bounds')
except ZeroDivisionError:
    print('Division by zero')

print('Execution terminated normally')
```

Output:

```
File "C:\Users\rooman\Anaconda3\lib\site-packages
\spyder_kernels\customize\spydercustomize.py", line 110,
in execfile
    exec(compile(f.read(), filename, 'exec'), namespace)

File "C:/Users/rooman/OneDrive/Desktop/python/test1.py",
line 10
    print(lst[num]/lst[den])
               ^
SyntaxError: default 'except:' must be last
```



Well that will give you an error, because when an exception object is generated it checks for except blocks sequentially therefore once the generic except block catches the exception, other specific handlers will never get chance to execute, therefore generic handlers are suppose be at the end

Next let us try to access the exception object and print its default message

```
print('Execution started normally')
lst=[10,20,0,40,50]
d={1:'c',2:'java',3:'python',4:'c++'}

try:
    r=int(input('Enter the rank of language:\n'))
    print(d[r])
    num=int(input('Enter the index of numerator:'))
    den=int(input('Enter the index of denominator:'))
    print(lst[num]/lst[den])
except KeyError as e:      #specific handlers
    print(e)
except IndexError as e:
    print(e)
except ZeroDivisionError as e:
    print(e)
except Exception as e:     #generic handler
    print(e)

print('Execution terminated normally')
```

Let us give the same inputs and check the message.

```
In [12]: runfile('C:/Users/rooman/OneDrive/Desktop/python/
test1.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
Execution started normally
```

```
Enter the rank of language:
5
5
Execution terminated normally
```

It just displays the key when keyerror is encountered.

```
In [13]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test1.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
Execution started normally

Enter the rank of language:
3
python

Enter the index of numerator:5

Enter the index of denominator:3
list index out of range
Execution terminated normally
```

Okay this is an appropriate message. Great!

```
In [14]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test1.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
Execution started normally
```

```
Enter the rank of language:
3
python
```

```
Enter the index of numerator:3
```

```
Enter the index of denominator:2
division by zero
Execution terminated normally
```

Appropriate message here as well



```
In [15]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test1.py', wdir='C:/Users/rooman/OneDrive/Desktop/python')
Execution started normally
```

```
Enter the rank of language:
five
invalid literal for int() with base 10: 'five'
Execution terminated normally
```

So we are getting appropriate messages. So it is up to the user which one they want to go with.

Python Fundamentals

day 63

Today's Agenda

- Exception handling contd



Exception handling contd

Let us see how exception handling mechanism works when multiple method calls are involved through the following example

```
def fun2():
    print('fun2() started execution')
    num=int(input('Enter the numerator'))
    den=int(input('Enter the denominator'))
    res=num/den
    print(res)
    print('fun2() finished execution normally')

def fun1():
    print('fun1() started execution')
    fun2()
    print('fun1() finished execution normally')

def main():
    print('main() started execution')
    fun1()
    print('main() finished execution normally')

main()
```

Output:

```
File "C:\Users\rooman\Anaconda3\lib\site-packages
\spyder_kernels\customize\spydercustomize.py", line 110,
in execfile
    exec(compile(f.read(), filename, 'exec'), namespace)

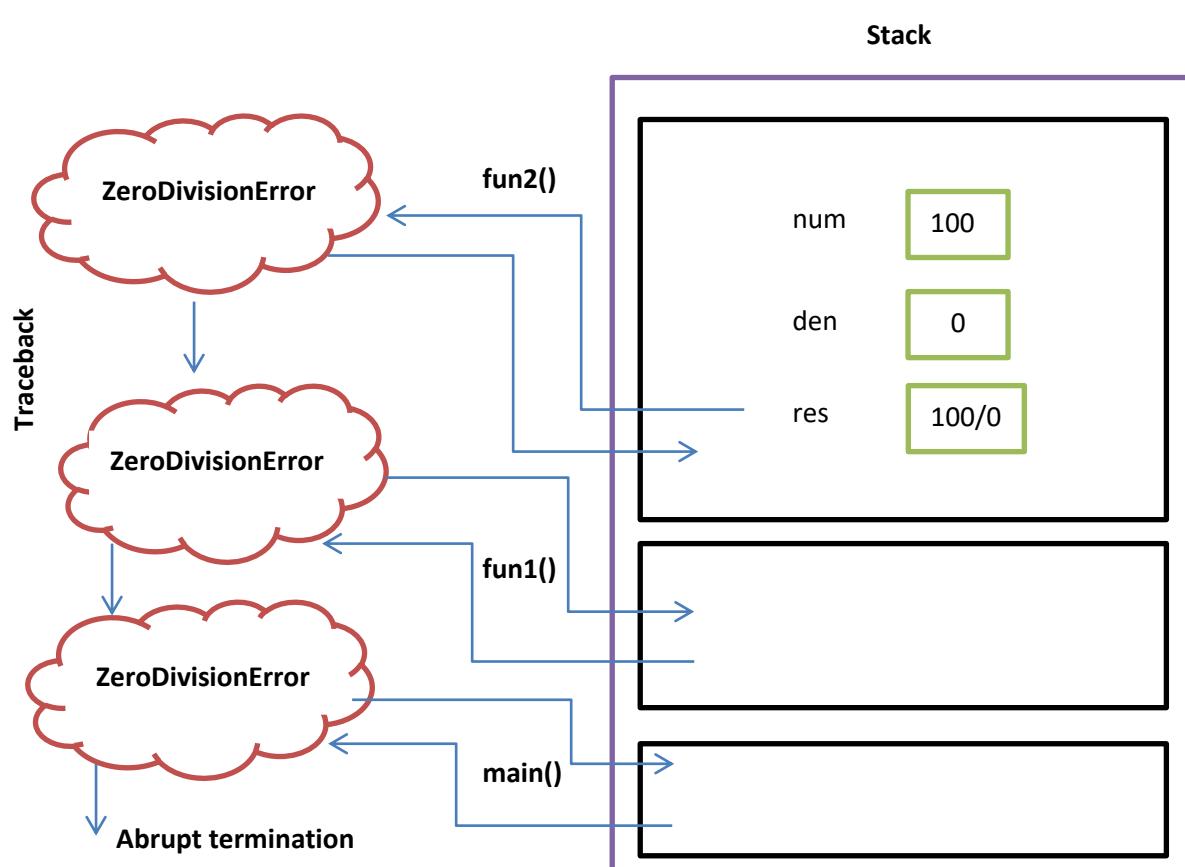
File "C:/Users/rooman/Downloads/test.py", line 19, in
<module>
    main()

File "C:/Users/rooman/Downloads/test.py", line 16, in
main
    fun1()

File "C:/Users/rooman/Downloads/test.py", line 11, in
fun1
    fun2()

File "C:/Users/rooman/Downloads/test.py", line 5, in
fun2
    res=num/den

ZeroDivisionError: division by zero
```



If except block is not in the method where exception is generated then it traces back to the method who called for it and checks in that method. If anywhere it is not present then the program is abruptly terminated.

Let us try to handle this exception

```
def fun2():
    print('fun2() started execution')
    try:
        num=int(input('Enter the numerator'))
        den=int(input('Enter the denominator'))
        res=num/den
        print(res)
    except ZeroDivisionError:
        print('Exception handled in fun2()')

    print('fun2() finished execution normally')

def fun1():
    print('fun1() started execution')
    fun2()
    print('fun1() finished execution normally')

def main():
    print('main() started execution')
    fun1()
    print('main() finished execution normally')

main()
```

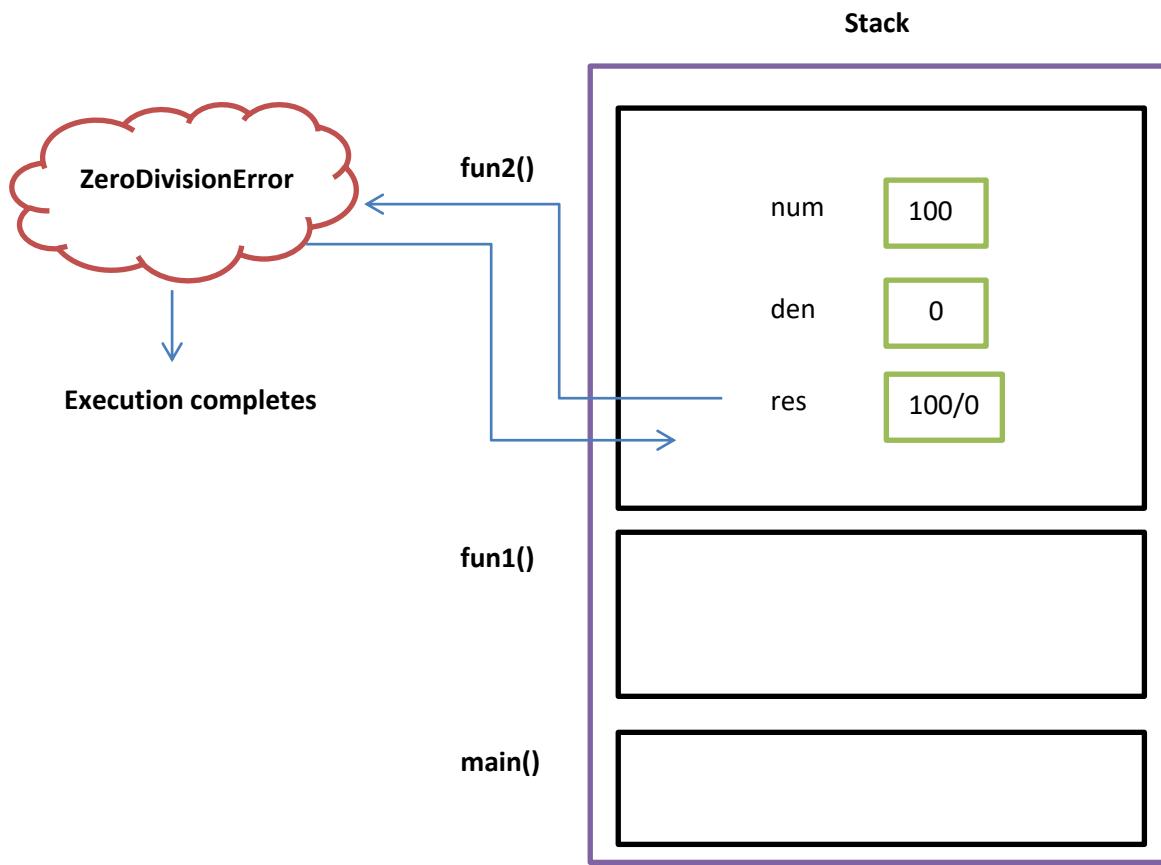
Output:

```
In [5]: runfile('C:/Users/rooman/Downloads/test.py',
wdir='C:/Users/rooman/Downloads')
main() started execution
fun1() started execution
fun2() started execution

Enter the numerator100

Enter the denominator0
Exception handled in fun2()
fun2() finished execution normally
fun1() finished execution normally
main() finished execution normally
```





Let us see what happens when the `except` block is in `fun1()`

```

def fun2():
    print('fun2() started execution')
    num=int(input('Enter the numerator'))
    den=int(input('Enter the denominator'))
    res=num/den
    print(res)
    print('fun2() finished execution normally')

def fun1():
    print('fun1() started execution')
    try:
        fun2()
    except ZeroDivisionError:
        print('Exception handled in fun1()')

    print('fun1() finished execution normally')

def main():
    print('main() started execution')
    fun1()
    print('main() finished execution normally')

main()

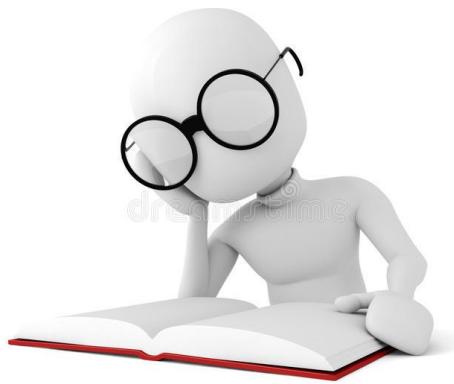
```

Output:

```
In [5]: runfile('C:/Users/rooman/Downloads/test.py',
wdir='C:/Users/rooman/Downloads')
main() started execution
fun1() started execution
fun2() started execution

Enter the numerator100

Enter the denominator0
Exception handled in fun2()
fun2() finished execution normally
fun1() finished execution normally
main() finished execution normally
```



We can see that exception is been handled, but fun2() did not finish it's execution as the control came to fun1() along with exception object.

Let us also try placing the except block in main()

```
def fun2():
    print('fun2() started execution')
    num=int(input('Enter the numerator'))
    den=int(input('Enter the denominator'))
    res=num/den
    print(res)
    print('fun2() finished execution normally')

def fun1():
    print('fun1() started execution')
    fun2()
    print('fun1() finished execution normally')

def main():
    print('main() started execution')
    try:
        fun1()
    except ZeroDivisionError:
        print('Exception handled in main()')

    print('main() finished execution normally')

main()
```

Output:

```
In [7]: runfile('C:/Users/rooman/Downloads/test.py',
wdir='C:/Users/rooman/Downloads')
main() started execution
fun1() started execution
fun2() started execution

Enter the numerator100

Enter the denominator0
Exception handled in main()
main() finished execution normally
```

As we see exception is handled but fun1() and fun2() did not finish their execution.



Python Fundamentals

day 64

Today's agenda

- Exception handling keywords



Exception handling keywords

During handling the exceptions, we came across few keywords such as try, except, else. But there are few more keywords you should be aware of like raise and finally.

Try: The try block lets you test a block of code for errors.

Except: The except block lets you handle the error.

Else: The else block is used to define the code to be executed if no errors were raised.

Let us see the other two keyword's functionality with the example below

Raise: As a python developer you can choose to throw an exception if the condition occurs. To throw (or raise) an exception, use the raise keyword. As shown below

```
def validate(mob):
    if len(mob)==10:
        print('Valid mobile number')
    else:
        raise ValueError

def main():
    mob=input()
    validate(mob)

main()
```



Output:

```
File "C:/Users/rooman/Downloads/test.py", line 11, in
<module>
    main()

File "C:/Users/rooman/Downloads/test.py", line 9, in
main
    validate(mob)

File "C:/Users/rooman/Downloads/test.py", line 5, in
validate
    raise ValueError

ValueError
```

In the above example we are trying to just raise a value error exception.

Let us see if it works fine with 10 digit input.

```
In [3]: runfile('C:/Users/rooman/Downloads/test.py',
kdir='C:/Users/rooman/Downloads')

9988776655
Valid mobile number          Activate Windows
```

We can see the code works perfectly well with 10 digit input.

Let us take another example where we are trying to raise NameError exception

```

def menu(item):
    if item=='pizza':
        print('Enjoy your pizza')
    elif item=='idli':
        print('Enjoy your idli')
    elif item=='burger':
        print('Enjoy your burger')
    else:
        raise NameError

def main():
    item=input()
    menu(item)

main()

```

Output:

```

File "C:/Users/rooman/Downloads/test.py", line 15, in <module>
  main()

File "C:/Users/rooman/Downloads/test.py", line 13, in main
  menu(item)

File "C:/Users/rooman/Downloads/test.py", line 9, in menu
  raise NameError

NameError

```

As the input given was pasta, and none of the condition is true else block is executed where we are trying to generate NameError exception.

```
In [5]: runfile('C:/Users/rooman/Downloads/test.py', wdir='C:/Users/rooman/Downloads')
```

```

idli
Enjoy your idli

```

Activate Windows

And it works completely fine with expected food items.

Let us next explore the finally keyword with an example

Finally: The finally block if specified will be executed regardless if the try block raises an error or not.



```

def fun():
    print('fun() started execution')

    try:
        num=int(input('Enter the numerator: '))
        den=int(input('Enter the denominator: '))
        res=num/den
        print(res)
    except ZeroDivisionError:
        print('Exception handled in fun()')

    print('fun() finished execution normally')

def main():
    print('main() started execution')
    fun()
    print('main() finished execution normally')

main()

```

Output:

```

In [6]: runfile('C:/Users/rooman/Downloads/test.py', wdir='C:/Users/
rooman/Downloads')
main() started execution
fun() started execution

Enter the numerator: 100

Enter the denominator: 2
50.0
fun() finished execution normally
main() finished execution normally

```

Activate Windows

In the above output we can see the code is working fine. Let us give another incorrect input and check the working flow

```

In [7]: runfile('C:/Users/rooman/Downloads/test.py', wdir='C:/Users/
rooman/Downloads')
main() started execution
fun() started execution

Enter the numerator: 100

Enter the denominator: 0
Exception handled in fun()
fun() finished execution normally
main() finished execution normally

```

We can notice the exception generated and handled in fun() but main() is unaware of the exception generated. Let us see below how to resolve this

```
def fun():
    print('fun() started execution')

    try:
        num=int(input('Enter the numerator: '))
        den=int(input('Enter the denominator: '))
        res=num/den
        print(res)
    except ZeroDivisionError as e:
        print('Exception handled in fun()')
        raise e

    print('fun() finished execution normally')

def main():
    print('main() started execution')
    try:
        fun()
    except:
        print('Exception reached main() has been handled')

    print('main() finished execution normally')

main()
```

Output:

```
In [9]: runfile('C:/Users/rooman/Downloads/test.py', wdir='C:/Users/
rooman/Downloads')
main() started execution
fun() started execution

Enter the numerator: 100

Enter the denominator: 0
Exception handled in fun()
Exception reached main() has been handled
main() finished execution normally
```

Activate Windows

In above output we can see main() has received the exception. But fun() has not completed its execution. Let's resolve this

```

def fun():
    print('fun() started execution')

    try:
        num=int(input('Enter the numerator: '))
        den=int(input('Enter the denominator: '))
        res=num/den
        print(res)
    except ZeroDivisionError as e:
        print('Exception handled in fun()')
        raise e
    finally:
        print('fun() finished execution normally')

def main():
    print('main() started execution')
    try:
        fun()
    except:
        print('Exception reached main() has been handled')

    print('main() finished execution normally')

main()

```

Output:

```

In [10]: runfile('C:/Users/rooman/Downloads/test.py', wdir='C:/Users/
rooman/Downloads')
main() started execution
fun() started execution

Enter the numerator: 100

Enter the denominator: 0
Exception handled in fun()
fun() finished execution normally
Exception reached main() has been handled
main() finished execution normally      Activate Windows

```

Now all the function are executing and are completing the execution process perfectly as expected.

At last let us consider all the blocks and see the flow of program.

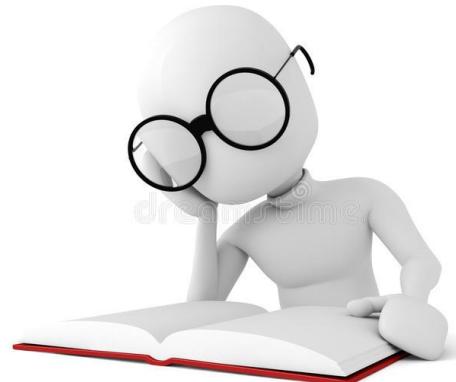
```

def fun(x):
    try:
        res=100/x
        print('Inside try')
    except:
        print('Inside except')
    else:
        print('Inside else')
    finally:
        print('Inside finally')

def main():
    x=int(input('Enter x:'))
    fun(x)

main()

```



Output:

```

In [11]: runfile('C:/Users/rooman/Downloads/test.py', wdir='C:/Users/
rooman/Downloads')

Enter x:2
Inside try
Inside else
Inside finally

```

As the given input is valid, except block will not execute and as no errors/exception were encountered else block will execute and then at last irrespective of the exception generated finally block will execute.

```

In [12]: runfile('C:/Users/rooman/Downloads/test.py', wdir='C:/Users/
rooman/Downloads')

Enter x:0
Inside except
Inside finally

```

Activate Windows

As 0 is the input, ZeroDivisionError exception object will be generated and except block is executed and then at the last finally block gets executed.

Python Fundamentals

day 65

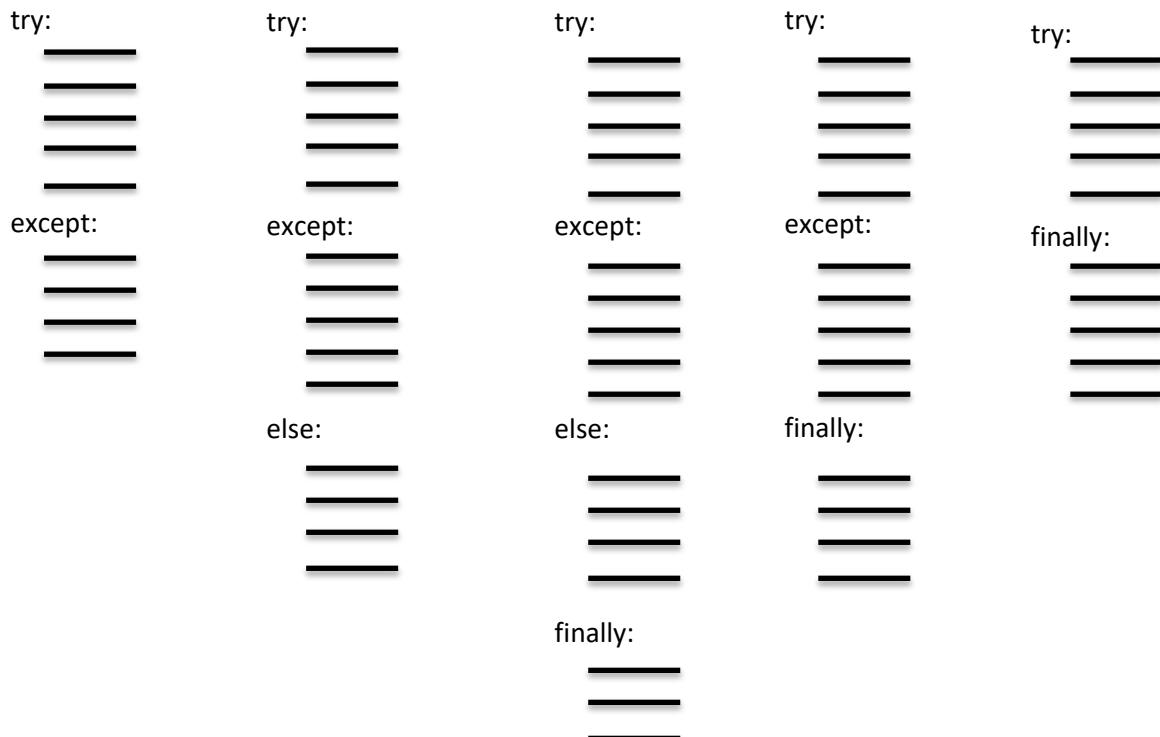
Today's agenda

- Valid combinations
- Exception hierarchy
- Custom Exceptions

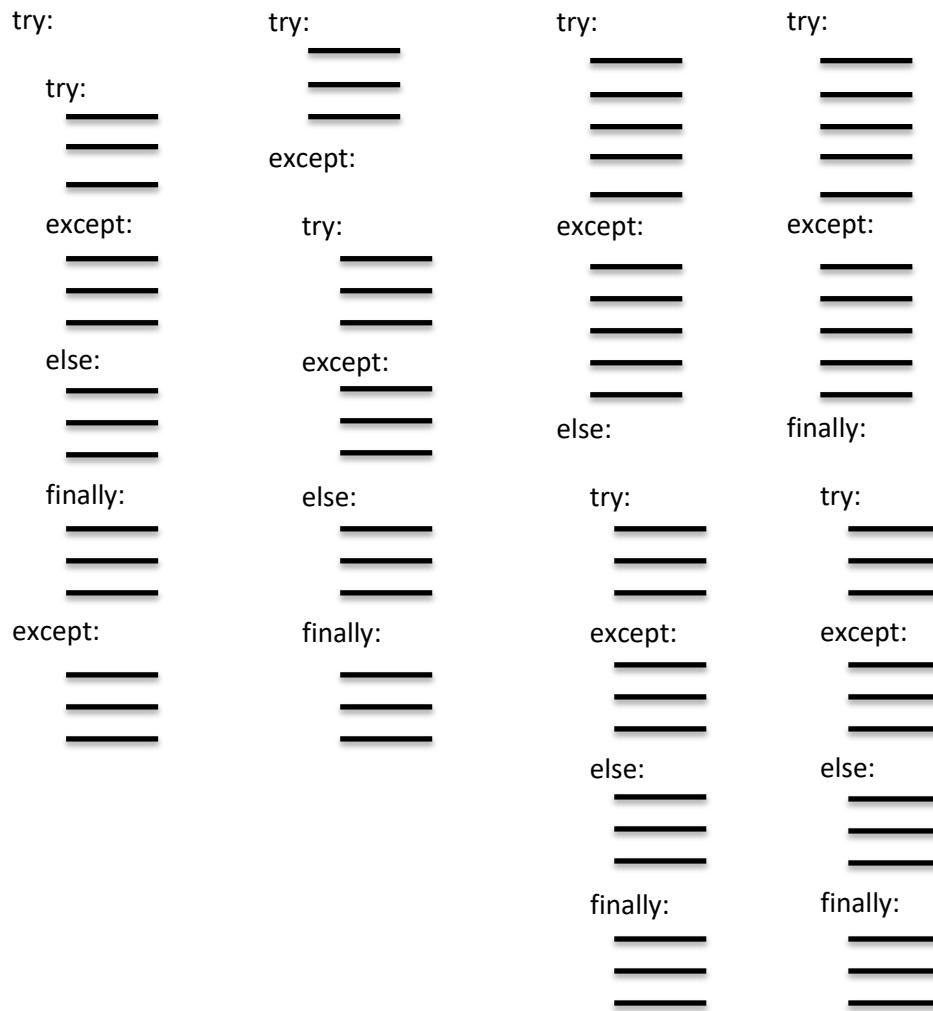


Valid Combinations

We have seen several blocks used in exception handling. Let us see what are these valid combinations of blocks in below segment



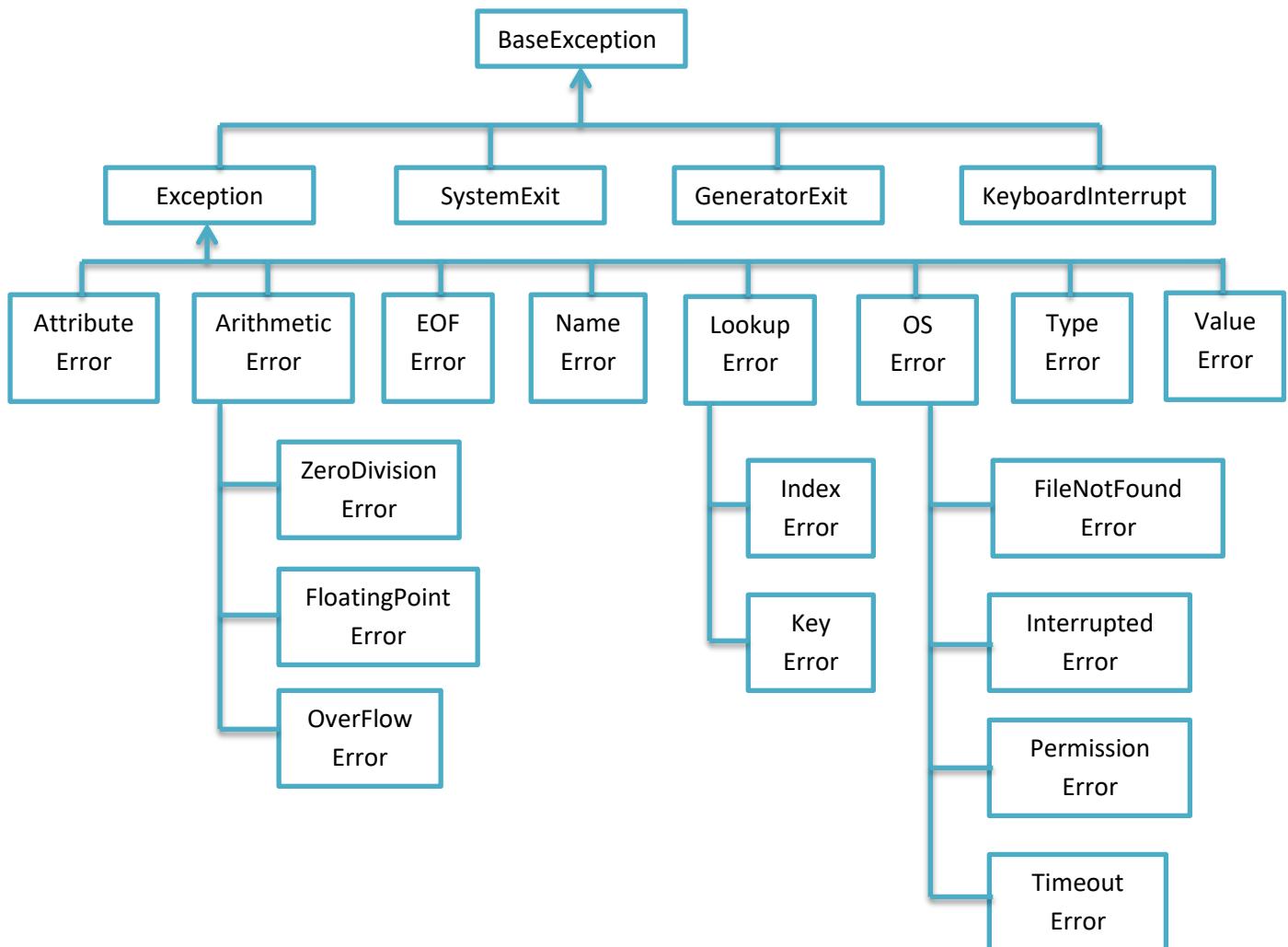
We can also have these nested in one another



Exception Hierarchy

The Python exception class hierarchy consists of a few dozen different exceptions spread across a handful of important base class types. In below chart let us see hierarchy in detail





Custom Exceptions

Let us previous example

```

def validate(mob):
    if len(mob)==10:
        print('Valid mobile number')
    else:
        raise ValueError

def main():
    mob=input()
    validate(mob)

main()
  
```



Output:

```
File "C:/Users/rooman/Downloads/test.py", line 11, in <module>
  main()

File "C:/Users/rooman/Downloads/test.py", line 9, in main
  validate(mob)

File "C:/Users/rooman/Downloads/test.py", line 5, in validate
  raise ValueError

ValueError
```

We can see we have got the value error. But it is not conveying proper message. The user might not understand why value error is generated. So to resolve this issue, in python users can create custom exceptions. Let us see how

```
class InvalidMobileNumberError(Exception):
    pass

def validate(mob):
    if len(mob)==10:
        print('Valid Mobile Number')
    else:
        raise InvalidMobileNumberError('Enter 10 digit mobile number')

def main():
    mob=input()
    validate(mob)

main()
```

Output:

```
File "C:/Users/rooman/Downloads/test.py", line 14, in <module>
  main()

File "C:/Users/rooman/Downloads/test.py", line 12, in main
  validate(mob)

File "C:/Users/rooman/Downloads/test.py", line 8, in validate
  raise InvalidMobileNumberError('Enter 10 digit mobile number')

InvalidMobileNumberError: Enter 10 digit mobile number
```

Now we can see exception generated clearly conveys the message.
Let us take another example and get more clarity

```
def menu(item):
    if item=='pizza':
        print('Enjoy your pizza')
    elif item=='idli':
        print('Enjoy your idli')
    elif item=='burger':
        print('Enjoy your burger')
    else:
        raise NameError

def main():
    item=input()
    menu(item)

main()
```



Output:

```
File "C:/Users/rooman/Downloads/test.py", line 15, in <module>
  main()

File "C:/Users/rooman/Downloads/test.py", line 13, in main
  menu(item)

File "C:/Users/rooman/Downloads/test.py", line 9, in menu
  raise NameError

NameError
```

Given pasta as the input we got NameError. But user may not know why is error generated, therefore lets customise the exception

```
class ItemNotInMenuError(Exception):
    pass

def menu(item):
    if item=='pizza':
        print('Enjoy your pizza')
    elif item=='idli':
        print('Enjoy your idli')
    elif item=='burger':
        print('Enjoy your burger')
    else:
        raise ItemNotInMenuError('Item not present in menu')

def main():
    item=input()
    menu(item)

main()
```

Output:

```
File "C:/Users/rooman/Downloads/test.py", line 18, in <module>
    main()

File "C:/Users/rooman/Downloads/test.py", line 16, in main
    menu(item)

File "C:/Users/rooman/Downloads/test.py", line 12, in menu
    raise ItemNotInMenuError('Item not present in menu')

ItemNotInMenuError: Item not present in menu
```

Now we can see the proper message and resolve it by giving try and except blocks

```
class ItemNotInMenuError(Exception):
    pass

def menu(item):
    if item=='pizza':
        print('Enjoy your pizza')
    elif item=='idli':
        print('Enjoy your idli')
    elif item=='burger':
        print('Enjoy your burger')
    else:
        raise ItemNotInMenuError('Item not present in menu')

def main():
    item=input()
    try:
        menu(item)
    except ItemNotInMenuError as e:
        print(e)

main()
```

Output:

```
In [17]: runfile('C:/Users/rooman/Downloads/test.py', wdir='C:/Users/rooman/Downloads')

pasta
Item not present in menu
```

Activate Windows

Now let us take an example of creating an account. Which will check if the username is unique and password satisfying the conditions

```
class DuplicateUserError(Exception):
    pass

class WeakPasswordError(Exception):
    pass

class User:

    user_name=set()

    def __init__(self,un,mob,pwd):
        self.un=un
        self.mob=mob
        self.pwd=pwd
        self.add_user()
        self.validate()

    def add_user(self):
        if self.un in User.user_name:
            raise DuplicateUserError('Username already exists')
        else:
            User.user_name.add(self.un)

    def validate(self):
        uc=lc=num=sp=0
        for i in self.pwd:
            if i.isupper():
                uc+=1
            elif i.islower():
                lc+=1
            elif i.isdigit():
                num+=1
            else:
                sp+=1

        if len(self.pwd)<6 or uc==0 or\
           lc==0 or num==0 or sp==0:
            raise WeakPasswordError('Password not strong enough')

    def main():
        un=input('Enter Username: ')
        mob=int(input('Enter Mobile: '))
        pwd=input('Enter the password: ')
        try:
            u1=User(un,mob,pwd)
            u2=User(un,mob,pwd)
        except DuplicateUserError as e:
            print(e)
        except WeakPasswordError as e:
            print(e)
        except:
            print('Hey some issue occurred')
        else:
            print('Account created successfully')

main()
```



Output:

```
In [18]: runfile('C:/Users/rooman/Downloads/test.py', wdir='C:/Users/rooman/Downloads')

Enter Username: pythonfan

Enter Mobile: 8899562325

Enter the password: #rooman123
Password not strong enough
```

Let us give existing username and check the output

```
In [19]: runfile('C:/Users/rooman/Downloads/test.py', wdir='C:/Users/rooman/Downloads')

Enter Username: pythonfan

Enter Mobile: 8899562325

Enter the password: #Roooman123
Username already exists
```

Activate Windows



Python Fundamentals

Day 66

Today's Agenda

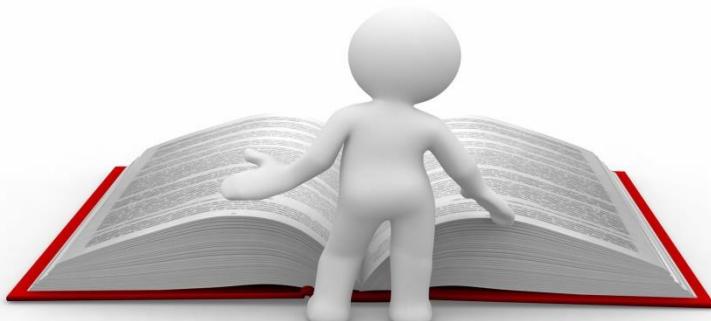
- Loggers
- Levels of logging
- Examples



Loggers

In simple words logger is a piece of code which the user will be attaching in the program. This logger will capture some information from your program and store it in a log file.

What is the information it is going to capture? Why should it be stored in a file? What is the advantage of doing this? Is what we shall see one by one. Let's start with one example that takes list of integers from user, the program will call a function which will take only even elements present in the list add them and return the sum.



```

def sum_even(lst):
    sum=0
    for i in lst:
        if i%2==0:
            sum=sum+i
    return sum

def main():
    lst=list(map(int,input().split()))
    res=sum_even(lst)
    print(res)
main()

```



Output:

```

In [2]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test1.py',
               wdir='C:/Users/rooman/OneDrive/Desktop/python')

5 6 7 8 9 10
24

```

There's nothing new in the above code, but let's trace it once

```

def sum_even(lst):
    print('sum_even() started execution')
    sum=0
    for i in lst:
        if i%2==0:
            sum=sum+i
    print('sum_even() finished execution')
    return sum

def main():
    print('main() started execution')
    lst=list(map(int,input().split()))
    print('input taken from the users')
    print('calling sum_even()')
    res=sum_even(lst)
    print('result of sum_even() collected')
    print(res)
    print('main() finished execution')

main()

```

Output:

```
In [3]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test1.py',  
wdir='C:/Users/rooman/OneDrive/Desktop/python')  
main() started execution  
  
5 6 7 8 9 10  
input taken from the users  
calling sum_even()  
sum_even() started execution  
sum_even() finished execution  
result of sum_even() collected  
24  
main() finished execution
```

We can see the above code has too much of information which is not important for the user or the client. Only if the programmer know the code and it's tracing it's enough. And using print statements everywhere is not a good habit of experienced programmer. To overcome this we have loggers where we can store the tracing of program and refer it whenever needed. Let us see how to achieve it

- Create a logger - let's start with basic one i.e. root logger.
- Connect the logger to a file (log file).
- Tell what information to be collected and stored in log file.



```

import logging

def sum_even(lst):
    logging.info('sum_even() started execution')
    sum=0
    for i in lst:
        if i%2==0:
            sum=sum+i
    logging.info('sum_even() finished execution')
    return sum

def main():
    logging.info('main() started execution')
    lst=list(map(int,input().split()))
    logging.info('input taken from the users')
    logging.info('calling sum_even()')
    res=sum_even(lst)
    logging.info('result of sum_even() collected')
    print(res)
    logging.info('main() finished execution')

main()

```

Output:

```
In [4]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test1.py',
              wdir='C:/Users/rooman/OneDrive/Desktop/python')
```

```
5 6 7 8 9 10
24
```

Now we can see no print statements is visible on output screen. Wondering how to access that information? Let us see

The information is stored in a certain order i.e.

<level> : <logger> : <message>

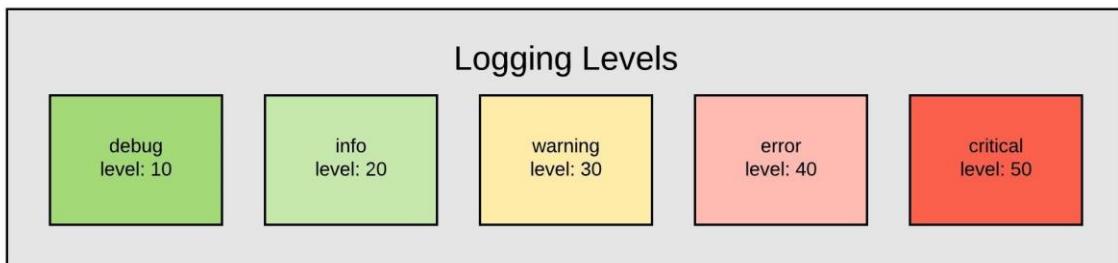
This information will be present in your hard disk where the program is saved.



```
*log1 - Notepad  
File Edit Format View Help  
INFO:root:main() started execution  
INFO:root:Input taken from user  
INFO:root:calling sum_even()  
INFO:root:sum_even() started execution  
INFO:root:sum_even() finished execution  
INFO:root:result of sum_even() collected  
INFO:root:main() finished execution
```

Levels of logging

We have several levels in logging which will define what data is being stored in the log file.



Debug: It is used when we want to store debugging related information in log file.

Info: It is used for tracing of program.

Warning: It is used to store warning related information.

Error: It is used whenever exception related information is to be stored.

Critical: It is used to capture information which results in critical failure of application.

Let us start exploring with examples

Example for debugging:

```
def add(x,y):
    return x+y

def sub(x,y):
    return x-y

def mul(x,y):
    return x*y

def div(x,y):
    return x/y

def main():
    a=10
    b=5
    print(f'a={a}')
    print(f'b={b}')
    res1=add(a,b)
    print(f'res1={res1}')
    res2=sub(a,b)
    print(f'res2={res2}')
    res3=mul(a,b)
    print(f'res3={res3}')
    res4=div(a,b)
    print(f'res4={res4}')

main()
```



Output:

```
In [6]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test1.py',
wdir='C:/Users/rooman/OneDrive/Desktop/python')
a=10
b=5
res1=15
res2=5
res3=50
res4=2.0
```

Activate Windows

Above is the normal code. Let us make changes and add logging features to it.

```
import logging

def add(x,y):
    return x+y

def sub(x,y):
    return x-y

def mul(x,y):
    return x*y

def div(x,y):
    return x/y

def main():
    logging.basicConfig(filename='log.txt', level=logging.DEBUG)
    a=int(input())
    b=int(input())
    logging.debug(f'a={a}')
    logging.debug(f'b={b}')
    res1=add(a,b)
    logging.debug(f'res1={res1}')
    res2=sub(a,b)
    logging.debug(f'res2={res2}')
    res3=mul(a,b)
    logging.debug(f'res3={res3}')
    res4=div(a,b)
    logging.debug(f'res4={res4}')

main()
```



Output:

```
In [7]: runfile('C:/Users/rooman/OneDrive/Desktop/python/test1.py',
               wdir='C:/Users/rooman/OneDrive/Desktop/python')
```

```
10
```

```
5
```

Activate Windows

Definitely all the results will be stored in log file. As shown below

log - Notepad
File Edit Format View Help

```
DEBUG:root:a=10
DEBUG:root:b=20
DEBUG:root:res1=30
DEBUG:root:res2=-10
DEBUG:root:res3=200
DEBUG:root:res4=0.5
```

Example for Warning:

```
def validate(num):
    if len(num)==10:
        print("Valid mobile number")
    else:
        print("Invalid mobile number")

def main():
    num=input("Enter mobile number:")
    validate(num)

main()
```



Output:

```
In [4]: runfile('C:/Users/rooman/OneDrive/Desktop/code/test1.py',
wdir='C:/Users/rooman/OneDrive/Desktop/code')
```

```
Enter mobile number:9988665577
Valid mobile number
```

```
In [5]: runfile('C:/Users/rooman/OneDrive/Desktop/code/test1.py',
wdir='C:/Users/rooman/OneDrive/Desktop/code')
```

```
Enter mobile number:559968709
Invalid mobile number
```

Activate Windows

Let us see how to log the warning message into the log file

```
import logging

logging.basicConfig(filename='log.txt', level=logging.WARNING)

def validate(num):
    if len(num)==10:
        print("Valid mobile number")
    else:
        logging.warning("Mobile number validation failed")
        print("Invalid mobile number")

def main():
    num=input("Enter mobile number:")
    validate(num)

main()
```

Output:

```
In [6]: runfile('C:/Users/rooman/OneDrive/Desktop/code/test1.py',
wdir='C:/Users/rooman/OneDrive/Desktop/code')
```

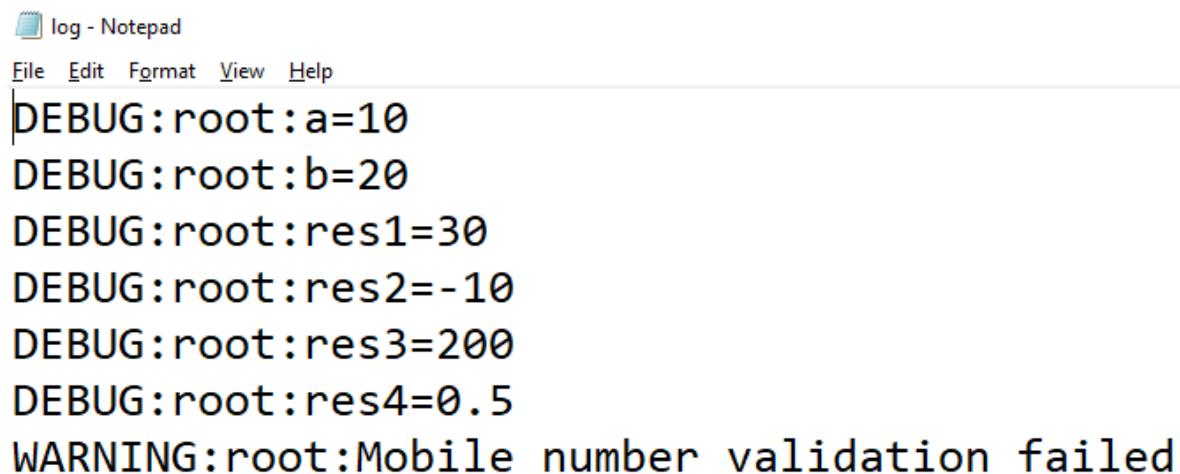
```
Enter mobile number:9988776655
Valid mobile number
```

```
In [7]: runfile('C:/Users/rooman/OneDrive/Desktop/code/test1.py',
wdir='C:/Users/rooman/OneDrive/Desktop/code')
```

```
Enter mobile number:998876544
Invalid mobile number
```

Activate Windows

Let us see if the log file has got updated log message



The screenshot shows a Notepad window titled "log - Notepad". The menu bar includes File, Edit, Format, View, and Help. The main content area displays the following log entries:

```
DEBUG:root:a=10
DEBUG:root:b=20
DEBUG:root:res1=30
DEBUG:root:res2=-10
DEBUG:root:res3=200
DEBUG:root:res4=0.5
WARNING:root:Mobile number validation failed
```

Certainly log file has got updated as expected.

Now let us see next example with **error** level

```
def div():
    num=int(input('Enter the numerator: '))
    den=int(input('Enter the denominator: '))
    q=num/den
    print(q)

def main():
    div()

main()
```



Output:

```
In [8]: runfile('C:/Users/rooman/OneDrive/Desktop/code/test1.py',
               wdir='C:/Users/rooman/OneDrive/Desktop/code')

Enter the numerator: 100

Enter the denominator: 2
50.0

File "C:/Users/rooman/OneDrive/Desktop/code/test1.py", line 8, in
main
    div()

File "C:/Users/rooman/OneDrive/Desktop/code/test1.py", line 4, in
div
    q=num/den

ZeroDivisionError: division by zero
```

Let us modify according with the concept of loggers

```
import logging

logging.basicConfig(filename='log.txt', level=logging.ERROR)

def div():
    try:
        num=int(input('Enter the numerator: '))
        den=int(input('Enter the denominator: '))
        q=num/den
        print(q)
    except:
        logging.error('Exception Occured')

def main():
    div()

main()
```

Output:

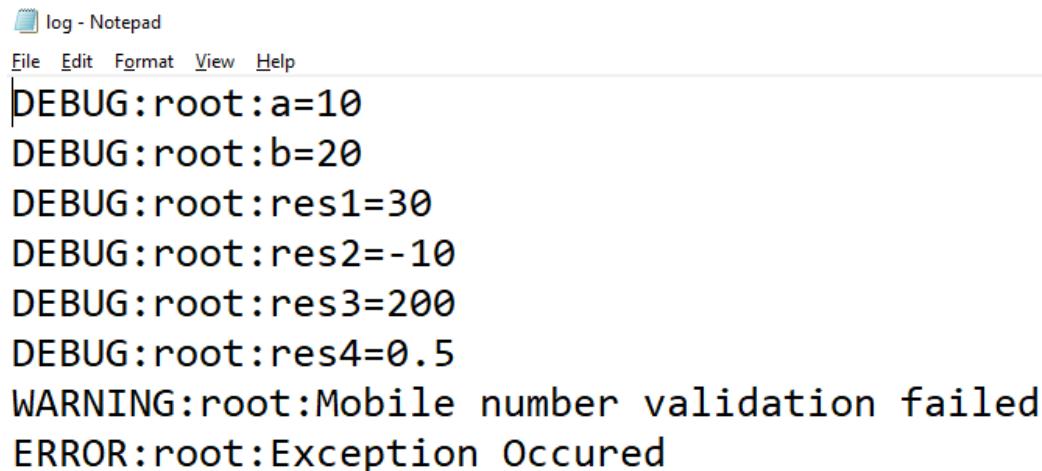
```
In [10]: runfile('C:/Users/rooman/OneDrive/Desktop/code/test1.py',  
      wdir='C:/Users/rooman/OneDrive/Desktop/code')
```

```
Enter the numerator: 100
```

```
Enter the denomenator: 0
```

Activate Windows

We can see that no error is displayed neither is the respective message. Let us check the log file to get the respective message.



Notepad window titled "log - Notepad" showing log entries:

```
File Edit Format View Help  
DEBUG:root:a=10  
DEBUG:root:b=20  
DEBUG:root:res1=30  
DEBUG:root:res2=-10  
DEBUG:root:res3=200  
DEBUG:root:res4=0.5  
WARNING:root:Mobile number validation failed  
ERROR:root:Exception Occured
```

Great!! We can see the message now, but is it conveying entire message? No. Let us see how to get the trace back of the exception occurred

```
import logging  
  
logging.basicConfig(filename='log.txt', level=logging.ERROR)  
  
def div():  
    try:  
        num=int(input('Enter the numerator: '))  
        den=int(input('Enter the denomenator: '))  
        q=num/den  
        print(q)  
    except:  
        logging.error('Exception Occured', exc_info=True)  
  
def main():  
    div()  
  
main()
```

Output:

```
In [11]: runfile('C:/Users/rooman/OneDrive/Desktop/code/test1.py',  
wdir='C:/Users/rooman/OneDrive/Desktop/code')
```

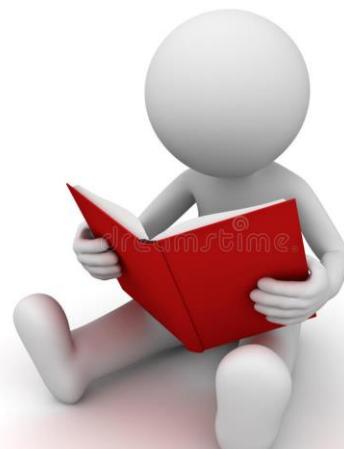
```
Enter the numerator: 100
```

```
Enter the denominator: 0
```

Let us check the log file and see what is the message we'll get after including exception info argument

```
log - Notepad  
File Edit Format View Help  
DEBUG:root:a=10  
DEBUG:root:b=20  
DEBUG:root:res1=30  
DEBUG:root:res2=-10  
DEBUG:root:res3=200  
DEBUG:root:res4=0.5  
WARNING:root:Mobile number validation failed  
ERROR:root:Exception Occured  
ERROR:root:Exception Occured  
ERROR:root:Exception Occured  
Traceback (most recent call last):  
  File "test1.py", line 9, in div  
    q=num/den  
ZeroDivisionError: division by zero
```

Great! We have received the trace back of the exception generated. But every single time the log message gets appended making it hard to know which message is appropriate one. Let us see how to change this, and get only the log message of that particular file executing.



```

import logging

logging.basicConfig(filename='log.txt', level=logging.ERROR, filemode='w')

def div():
    try:
        num=int(input('Enter the numerator: '))
        den=int(input('Enter the denominator: '))
        q=num/den
        print(q)
    except:
        logging.error('Exception Occured', exc_info=True)

def main():
    div()

main()

```

Output:

Output will be same as previous ones. Let us directly take a look at log file

*log - Notepad
File Edit Format View Help

```

ERROR:root:Exception Occured
Traceback (most recent call last):
  File "test1.py", line 9, in div
    q=num/den
ZeroDivisionError: division by zero

```

Awesome! We got it as expected. Here by default the filemode will be append(a), but we have changed it to write mode(w) and executed.

