Python revision

##Arithmetic Expressions: Precedence

Operators	Operation
`**`	exponentiation
`+x, -x`	unary +, unary - (positive sign), (negative sign)
`*, /, //, %`	multiplication, division, floor division, modulus
`+, -`	addition, subtraction

high

low

- In Python, multiplication *, division /, floor division //, and modulus % all have the same precedence level.
- When multiple operators of the same precedence appear in an expression, Python evaluates them from left to right (this is called left-to-right associativity).
- There are two exceptions to this rule, the ** and = operator, both of which are evaluated from right to left.

```
print(3 - 2 + 1)
print( 8 % 4 % 2)
print(2 ** 3 ** 0)
```

```
2
0
2
```

##Type of Expressions

1. Arithmetic
2. Binary

```
print(type(1 + 2))
```

```
<class 'int'>
```

##Integral Number to binary(i.e, 13)

- Number ko 2 se divide karo
- Har baar remainder likho (0 ya 1)
- Jab tak quotient 0 na ho jaaye
- Remainders ko ulṭe order mein likho (bottom to top)

Step	Division	Quotient	Remainder
1	13 ÷ 2	6	1
2	6 ÷ 2	3	0
3	3 ÷ 2	1	1
4	1 ÷ 2	0	1

✅ Final Binary: **1101**

(Remember: bottom se top padho → `1 1 0 1`)

##Floating number to binary(i.e, 0.625)

1. Decimal part ko 2 se multiply karo
2. Jo **integer part** mile, likh lo (0 ya 1)
3. Bachi hui **decimal part ko phir se multiply** karo
4. Jab tak 0 ya repeat na ho jaaye

🎯 Example: `0.625` ko binary mein badlo

Step	Multiply by 2	Result	Integer Part
1	0.625 × 2	1.25	1
2	0.25 × 2	0.5	0
3	0.5 × 2	1.0	1

✅ Final Binary: **0.101**

###13.625 ka binary representation hai: 1101.101

Binary to Number

Numbering starts from *Right to left*

- Binary of Fractional part *(+ve numbering)*: Numbering starts from 1
- Binary of Decimal part *(-ve numbering)*: Numbering starts from 0

1. Integer Part (Binary se Decimal)

Jab aap binary number ko decimal mein convert karte ho, integer part ko **positional values** ke basis par calculate karte hain.

Example: 1101 (binary) ko decimal mein convert karte hain:

- Start from rightmost digit (least significant bit) aur position-wise multiply karte hain powers of 2 se.
- **1101 (binary)**:
  - $1 \times 2^3 = 8$
  - $1 \times 2^2 = 4$
  - $0 \times 2^1 = 0$
  - $1 \times 2^0 = 1$

Decimal mein: 8 + 4 + 0 + 1 = **13**.

2. Fractional Part (Binary se Decimal)

Fractional part ko convert karte waqt, hum powers of 2 ka use karte hain, but **negative powers of 2**.

Example: .101 (binary) ko decimal mein convert karte hain:

- **.101** ka matlab hai:
  - $1 \times 2^{-1} = 0.5$
  - $0 \times 2^{-2} = 0$
  - $1 \times 2^{-3} = 0.125$

Decimal mein: 0.5 + 0 + 0.125 = **0.625**

Final Conversion (Binary to Decimal):

Agar aapke paas **binary number** hai **1101.101**, toh:

- Integer part (1101) = 13
- Fractional part (.101) = 0.625

Final decimal number: **13.625**

- Most of the number in binary can not be written exactly,
- For example, since computer store every thing in binary. Computer can not store exact value of 0.1 because 0.1(meaning 1/10) is a infinite representations in binary (0.00011001100110011001100110011... yeh pattern repeat hota rahega).
- like we can not write exact value of 1/3 in decimal form, we`ve to approx it as 0.33333...upto whatever decimal place.
- Trick: Agar decimal number 1 / (2^n) jaisa hai (jaise 0.5 = 1/2, 0.25 = 1/4), to wo binary mein exact likha ja sakta hai.

so a computer can store a floating number in maximum 53 bits of memory ?

Great question! The answer is:

✅ Yes, but with more detail:

A **floating-point number** (like 0.1, 3.14, etc.) is stored in memory using the **IEEE 754 standard** (which almost all modern computers follow).
In this standard, a **double-precision** floating-point number (commonly used in Python, C, Java, etc.) uses:

🧮 64 bits total, divided like this:

Part	Bits Used	Purpose
Sign	1 bit	Positive or Negative (0 or 1)
Exponent	11 bits	Determines how big/small number is (like powers of 2)
Mantissa	52 bits	Stores actual digits (also called "significand" or "fraction")

👇 But why do we say "53 bits of precision"?

Because there's an extra trick:

- The **mantissa (52 bits)** stores the fraction part
- But it's assumed to always have a leading **1** at the start (called the **hidden bit** or **implicit 1**)

➡️ So total **significant digits** become:

> **1 (hidden bit) + 52 bits = 53 bits of precision**

🎯 So yes:

A computer can **accurately store up to 53 binary digits** (bits) of precision in a double-precision float.
Beyond that? It has to **round off** the extra bits, which can cause small errors — like `0.1 + 0.2 = 0.30000000000000004`.

hidden bit is applied only when the number is a normalized number.

- A normalized number is one where the binary representation is written in the form 1.xxxxxxxxx * 2^n
- here, 1 - hidden bit(1 bit), xxxxxxxx - mantissa(52 bits), 2^n - Exponent(11 bit).

hidden bit is not used for the mumbers which can not be normalised (zero, subnormals) .

🧮 Example:

Let's say we want to store:

```php
6.25 → in binary: 110.01 = 1.1001 × 2^2
```

The normalized mantissa is:

```
1.1001
```

We **don't store** the leading `1.` — only `1001...` gets stored.

But whenever the computer reads it, it **adds that hidden 1 back** automatically.

🎯 Benefits:

Advantage	Explanation
💾 **Memory Efficient**	Saves 1 extra bit per float number
🎯 **Full Precision**	Still gives 53 bits of actual binary precision
🔄 **Automatic Handling**	The hardware/processor takes care of adding the 1 — programmer ko kuch nahi karna padta

So coming back to the point, **Beware of float!**

- machine round off or cut off all the binary number more than 53 bits of space & when this rounded off/new binary is converted back to the number it is slightly greater than the actual floating number.
- Example, 0.1 in binary ≈ 0.0001100110011001100110011001100110011... (repeat ho raha hai)
- 0.1 ≈ 0.0001100110011001100110011001100110011001100110011001101 (only 53 bits) ------->>>> 0.1000000000000000055511151231257827021181583404541015625 (slightly greater than 0.1)

```python
print(0.1 * 3)    # It gave 0.30000000000000004 instead of 0.3 (slightly greater than 0.3)c
```

```
0.30000000000000004
```

Python, and programming languages in general, do not support arbitrary precision for representing real numbers. When the number cannot be represented exactly, an approximate value is returned. As a result of this behaviour, we should be careful when using float values in expressions that involve comparisons.

```
print(10.00000000000000000000001 > 10)

False
```

The following expression presents a typical case of approximation when dealing with float. The number 0.1 ** 1000 is extremely small. So, the interpreter is going to represent that as 0

```
print(0.1 ** 100 == 0.0)
print(0.1 ** 1000 == 0.0)
print(0.1 * 3 == 0.3)

False
True
False
```

📌 Solution kya hai?

1. **Maths mein samajh lo ki yeh floating point ki limitation hai.**
2. Agar aapko precise decimal chahiye (jaise paisa ya banking), toh use karo:
   - Python ka `decimal` module
   - Ya phir round kar lo manually

```
from decimal import Decimal
print(Decimal('0.1') * 3 == 0.3)
print(Decimal('0.1') + Decimal('0.2'))  # Output: 0.3

False
0.3
```

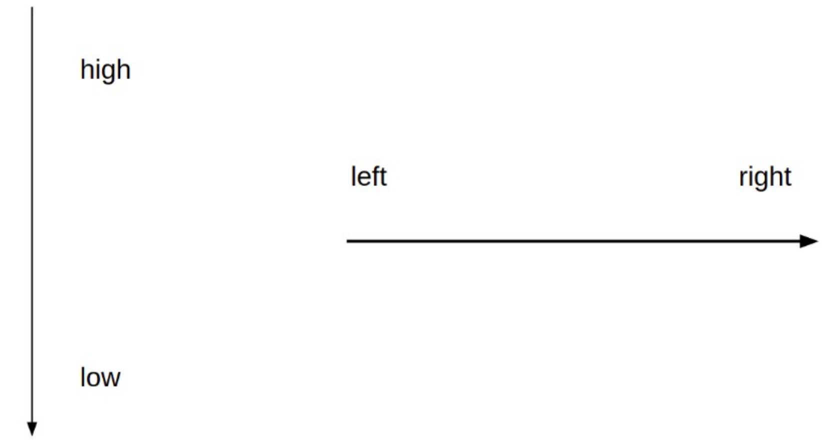
###What's happening?

- Decimal('0.1') * 3 results in a precise decimal value: Decimal('0.3')
- 0.3 on the right-hand side of the comparison is a floating-point number, not a Decimal.

Precedence and Order

Similar to arithmetic operators, logical operators also have precedence. Boolean expressions are also going to evaluated from left to right:

Operators
not
and
or




```
print(not True and False)  # (not True) and False ---> False
```

```
False
```

##Short Circuit Evaluation

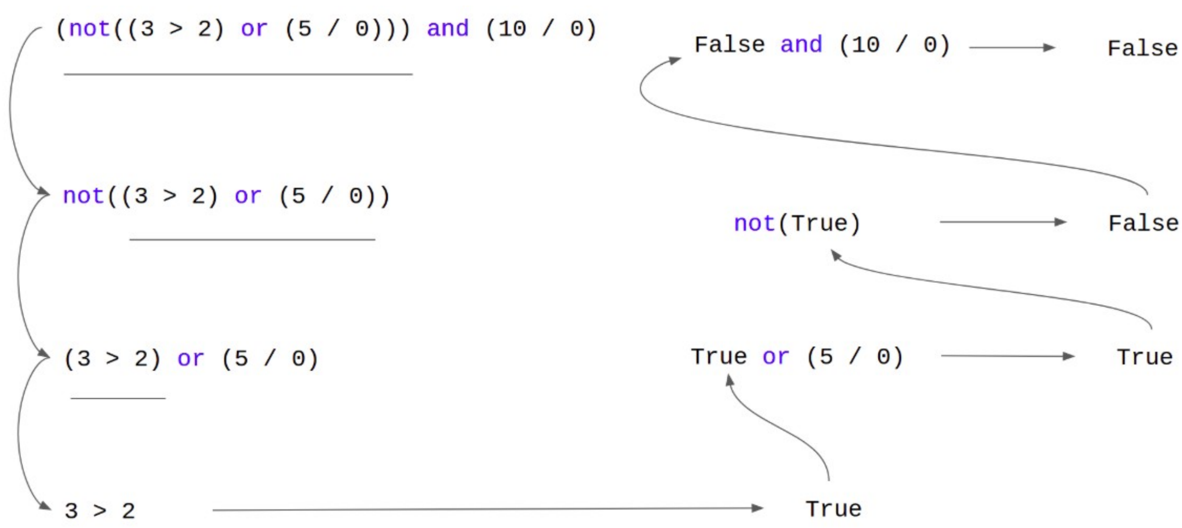
The expression is evaluated from left to right. The operator is or. Since the operand on the left is True, the whole expression will evaluate to True irrespective of the operand on the right. So, the interpreter skips evaluating the operand on the right. This behaviour is called short circuit evaluation.

```
True or (1 / 0)  #but 1/0 should give error ZeroDivisionError: division by zero
```

```
True
```

```
(not((3 > 2) or (5 / 0))) and (10 / 0)
```

```
False
```



##Errors

- Syntax Errors : error occured due to wrong syntex
- Exceptions : *runtime errors* — they happen after the code is syntactically correct, but something goes wrong while the program is running.

##Strings

- singe(' ') & double(" ") quote : Single line
- Triple quote (''' ''') : Multiple line
- Replication : `print("amiT" * 5)` ------> `amiTamiTamiTamiTamiT`
- Length of a string using the len function
- Concatenation of two strings using the + operator
- Indexing : from start (+ve indexing, starts from 0) & from end (-ve indexing, starts from -1)
- Slicing : substring = `string[start : end : step]`, *Note: step will include start indexed chr.*
- Immutability :
- We say that something is "mutable" if it can be changed, modified. Therefore, an object is immutable if it cannot be changed or modified. Strings are `immutable`. One or more characters in the string literal present in *word* cannot be modified in-place.
- `word = 'some string'`
- `word[0] = 'S'` ---------> *Error: TypeError : 'str' object does not support item assignment*

- Comparison : It uses alphabetical ordering *(like our english dictonary)* to compare two strings *(lexicographic ordering)*.
- Python compares characters based on Unicode code points.
- Decimal and hexadecimal are just different ways of viewing those code points
- Unicode formate : U+XXXX, here XXXX is the hexadecimal representation.
- Decimal value: It's the base-10 (decimal) number representing the character in Unicode or ASCII. i.e, 65 = U+0041, since 0041 is in hexadecimal, converting hexadecimal to decimal (base 10), we multiply each digit of the hex number by powers of 16. So, 65 = U+0041 = $0 \times 16^3 + 0 \times 16^2 + 4 \times 16^1 + 1 \times 16^0 = 64 + 1 = 65$ *the hexadecimal number 1123 is equal to 4387 in decimal:* example, 0041 of U+0041 is same as 0x41 in hexadecimal* $ 1123_{16} = (1 \times 16^3) + (1 \times 16^2) + (2 \times 16^1) + (3 \times 16^0) = 4387$
- Hexadecimal *(base-16 (hex) and, Prefix `0x` is used to indicate that the number is in hexadecimal)*: example, 0041 of U+0041 = 0x41 = $4 \times 16 + 1 = 65$
- The backslash `\` is called the escape character in python.
- `\n`for newline
- `\t` for tab
- A string is a substring of another string if the first string is contained in the second : Example, `good` is a substring of `verry good` string.

Methods

- It is a function defined in a class. *The `capitalize()` method in Python is associated with the `str class`* (i.e., string objects).*
- Called on an object using dot notation. example, `text.upper()`

✅ Your Code:

python

```python
text = "hello world"
print(text.capitalize())
```

🔍 What's Happening:

✅ 1. `"hello world"` is a string literal

- Python automatically treats it as an **object of class `str`**.

✅ 2. `text` becomes a reference to that `str` object:

python

```python
text = "hello world"  # → text is now an instance (object) of str
```

✅ 3. You can now call any method defined in the `str` class:

- `text.capitalize()` ✅
- `text.upper()` ✅
- `text.lower()` ✅
- `text.split()` ✅
- `text.replace()` ✅
- ... and many more!

✅ What Are **Keywords** in Programming?

📌 **Keywords are reserved words that have a special meaning in a programming language.**

- You **cannot use them as variable names**, function names, or identifiers.
- They are **built-in** to the language and form its **syntax and structure**.

🔹 Example in Python:

```python
if = 5      # ❌ This will cause an error
```

Here, `if` is a **keyword**, so Python will throw a syntax error if you try to use it as a variable name.

📚 Common Python Keywords (a few):

Keyword	Purpose
`if`	Conditional statement
`else`	Alternative branch
`for`	Looping
`while`	Looping

###Deleting Variables

- Variables can be deleted by using `del` keyword.

```python
x = 15
y = 16
print(x, y)
del x
#print(x, y)   #Try running this code, You`ll get NameError
```

```
15 16
```

###Input & conversion

```python
x = input()   # it takes str datatype by default
print(type(x))
```

```
x
<class 'str'>
```

```
y = int(input())    # we can convert the datatype from one to another using keyword line int(), str(), dict(), bool(), tuple(), set() etc.
print(type(y))

1
<class 'int'>
```

###Built-in Functions

- These are functions that have already been defined & can be called directly using name.
- Example, print(), type(), len(), sorted() etc.

```
pow(2,3,2) # pow(x, y, z) returns the value of x^y mod z. i.e, (x**y)%z   , % - gives remender wheres // gives quotient.

0
```

###Conditional Statements

- All independet `if blocks` will be checked & run accordingly.
- only one block will be run in case of `if-elif-else chain`. any of the condition which is true will execute & skip rest every block of this chain.
- elif is used when we want to check another condition only if the previous if condition was false.
- If we use multiple if statements instead of elif, Python would check all conditions, even when one is already true.

This would print multiple grades, which you don't want — it's incorrect logic for mutually exclusive conditions.

```
marks = int(input())

if marks >= 90:
    print("Grade: A")
if marks >= 80:
    print("Grade: B")
if marks >= 70:
    print("Grade: C")

100
Grade: A
Grade: B
Grade: C
```

What’s going on?

- Python checks conditions top to bottom.

- Once it finds a condition that’s True, it executes that block and skips the rest.
- So even if multiple conditions are true, only the first matching one will run.

```
marks = int(input("Enter your marks: "))

if marks >= 90:
    print("Grade: A")
elif marks >= 80:
    print("Grade: B")
elif marks >= 70:
    print("Grade: C")
elif marks >= 60:
    print("Grade: D")
else:
    print("Grade: F")
```

```
Enter your marks: 100
Grade: A
```

```
x = 25

if x < 10:
    print("Less than 10")
elif x < 20:
    print("Between 10 and 19")
elif x < 30:
  print("Between 20 and 29")
else:
    print("30 or more")
```

```
Between 20 and 29
```

Library

- calendar

```
import calendar
#print(calendar.prmonth(30000, 8))   #try running this it`ll give none additionally because calendar.prmonth(year, month) prints the calander but not returns anything that is why.
calendar.prmonth(1998, 12)
```

```
   December 1998
Mo Tu We Th Fr Sa Su
    1  2  3  4  5  6
 7  8  9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30 31
```

```
import calendar
calendar.weekday(1998, 12, 7)   # output: 0    meaning: Monday

0

import this             #These are some nuggets of wisdom from Tim
Peters, a "major contributor to the Python programming language"

import time
print(time.ctime())    #It prints the current time.

Mon Apr 21 09:26:37 2025
```

Loops

- while
- for

while

```
# It will take 58 days to write 1_000_000 numbers manualy.

num = 1_000_000     # _ in a number is used when we have large
numbers; improves readability
avg_time = 5         # if it takes 5 second to print a number, example
: print(5)
seconds = num * avg_time
minutes = seconds / 60
hours = minutes / 60
days = hours / 24
print('Approximate number of days =', round(days))

Approximate number of days = 58

total = 0
x = int(input())
while x > 0:
  total += x
  x = int(input())
print(total)

-1
0
```

For

- In each iteration of the loop, an element in the sequence is picked up and is printed to the console.
- Assuming that the sequence is ordered from left to right, the leftmost element is the first to be picked up.

- The sequence is processed from left to right.
- Once the rightmost element has been printed to the console, control returns to line-1 for one last time. Since there are no more elements to be read in the sequence, the control exits the loop and moves to line-3.

```
print(range(5))   # range is an object that represents a sequence of numbers starts from 0.
print(range(10,25, 3))  # range(start, end, step) Note: end not included.
type(range(5))
```

```
range(0, 5)
range(10, 25, 3)
```

```
range
```

```
range(5)
```

```
range(0, 5)
```

```
# for & in are the keywords in python.
for num in range(5):      # range(+ve integer sequence)
  print(num)
```

```
0
1
2
3
4
```

```
for num in range(1,10):              # range(start, end)
  print(num)
```

```
1
2
3
4
5
6
7
8
9
```

```
for num in range(1, 10, 2):          #range(start, end, step)
  print(num)
```

```
1
3
5
7
9
```

Loop in string

- Since a string is a sequence of characters, we can use the for/while loop to iterate through strings.

```
word = "Beautiful"
print(f'Length of word is {len(word)}')

index = 0
for chr in word:
  print(f'{chr} is indexed at {index} in the word {word}.')
  index += 1

Length of word is 9
B is indexed at 0 in the word Beautiful.
e is indexed at 1 in the word Beautiful.
a is indexed at 2 in the word Beautiful.
u is indexed at 3 in the word Beautiful.
t is indexed at 4 in the word Beautiful.
i is indexed at 5 in the word Beautiful.
f is indexed at 6 in the word Beautiful.
u is indexed at 7 in the word Beautiful.
l is indexed at 8 in the word Beautiful.

word = "Beautiful"
print(f'Length of word is {len(word)}')

index = 0
while index < len(word):
  print('{} is indexed at {} in the word
{}.'.format(word[index],index,word))
  index +=1


num = 2.345678896
print(f'num upto two decimal place is {num: .2f}')

Length of word is 9
B is indexed at 0 in the word Beautiful.
e is indexed at 1 in the word Beautiful.
a is indexed at 2 in the word Beautiful.
u is indexed at 3 in the word Beautiful.
t is indexed at 4 in the word Beautiful.
i is indexed at 5 in the word Beautiful.
f is indexed at 6 in the word Beautiful.
u is indexed at 7 in the word Beautiful.
l is indexed at 8 in the word Beautiful.
num upto two decimal place is  2.35
```

Break Continue

```python
# Write the least +ve number divisible by 2, 3 & 4.

num=1
while num:
  if (num%2 == 0) and (num%3 == 0) and (num%4 == 0):
    print(num)
    break
  num += 1
```

```
12
```

```python
# Print all the number divisible by 3 which are less than 50.

num = 1
while num < 50:
  num += 1
  if num % 3 == 0:
    print(num)
    continue
```

```
3
6
9
12
15
18
21
24
27
30
33
36
39
42
45
48
```

Nested loops

```python
#Find the number of ordered pairs of positive integers whose product
is 100. Note that order matters: (2, 50) and (50, 2) are two different
pairs.

count = 0
for x in range(1,101):
  for y in range(1,101):
    if x*y == 100:
      count +=1
print(count)
```

```
9
```

```python
import math
print(math.ceil(134.75))
```

```
135
```

```python
import math
print(math.floor(134.75))
```

```
134
```

```python
int(134.23)
```

```
134
```

If x is not a prime, it means it has at least one factor other than 1 and itself.

Let’s say x = a × b.

If both a and b were greater than √x, then: `a × b > √x × √x = x`

But that's a contradiction! So, at least one of the two numbers a or b must be ≤ √x.

so we have to check atleast one number less than the square root of the given number. If any is multiple of the given number tha it is not a prime number.

```python
#Find the number of prime numbers less than n, where n is some positive integer.

import math

num = int(input())

count = 0
for x in range(2, num):
  flag = True
  for y in range(2, int(math.sqrt(num)) + 1):    # we could have used just int(num * 0.5)
    if num %y == 0:
      flag = False
      break
  if flag:
    count += 1
print(count)
```

```
10
0
```

print: end, sep

- end

```
Whenever we use the print function, it prints the expression
passed to it and immediately follows it up by printing a newline.
This is the default behaviour of print. It can be altered by
using a special argument called end. The default value of end is
set to the newline character. So, whenever the end argument is
not explicitly specified in the print function, a newline is
appended to the input expression by default. In the code given
above, by setting end to be a comma, we are forcing the print
function to insert a comma instead of a newline at the end of the
expression passed to it. It is called end because it is added at
the end.
```

- sep

```
If multiple expressions are passed to the print function, it
prints all of them in the same line, by adding a space between
adjacent expressions. we can use "sep" to use some other
separator.
```

```
#Accept a positive integer n as input and print all the numbers from 1
to n in a single line separated by commas.

#Wrong way
num = int(input())
for x in range(1, num+1):
  print(x, end=',')

23
1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,

# Right way
num = int(input())
for x in range(1, num):
  print(x, end=',')
print(num)

21
1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21

print('amit', 'kumar', 'pandey')
print('amit', 'kumar', 'pandey', sep = ' | ')

amit kumar pandey
amit | kumar | pandey

#Accept a positive integer n, which is also a multiple of 3, as input
and print the following pattern:
# |1,2,3|4,5,6|7,8,9|...|n - 2,n - 1,n|

n = int(input())
```

```
print('|', end = '')
for i in range(1,n, 3):
  print(i, i+1, i+2, sep=',', end='|')
```

```
24
|1,2,3|4,5,6|7,8,9|10,11,12|13,14,15|16,17,18|19,20,21|22,23,24|
```

`10.2f`

- `.` represents decimal
- `2` represents round off upto two decimal place.
- `f` represents it is a floating point number.
- `10` represents the column space, atlease 10 chracter of space.

```
roll_1, marks_1 = 'BSC1001', 90.5
roll_2, marks_2 = 'BSC1002', 100
roll_3, marks_3 = 'BSC1003', 90.15
print(f'{roll_1}: {marks_1:10.2f}')
print(f'{roll_2}: {marks_2:10.2f}')
print(f'{roll_3}: {marks_3:10.2f}')
```

```
BSC1001:      90.50
BSC1002:     100.00
BSC1003:      90.15
```

```
roll_1, marks_1 = 'BSC1001', 90.5
roll_2, marks_2 = 'BSC1002', 100
roll_3, marks_3 = 'BSC1003', 90.15
print(f'{roll_1}: {marks_1:5.2f}')
print(f'{roll_2}: {marks_2:5.2f}')
print(f'{roll_3}: {marks_3:5.2f}')
```

```
BSC1001: 90.50
BSC1002: 100.00
BSC1003: 90.15
```

Goal : when iteration ---> n, value ---> limiting value

- meaning, find the limit.

n	x_n	Approximate value
1	$\sqrt{2}$	1.414
2	$\sqrt{2+\sqrt{2}}$	1.848
3	$\sqrt{2+\sqrt{2+\sqrt{2}}}$	1.962
4	$\sqrt{2+\sqrt{2+\sqrt{2+\sqrt{2}}}}$	1.990
5	$\sqrt{2+\sqrt{2+\sqrt{2+\sqrt{2+\sqrt{2}}}}}$	1.998

Isn't that beautiful? It looks like this sequence — the train of square roots — is approaching the value 2. Let us run the loop for more number of iterations this time:

```
# example of limit : In 20 itrations it is close to 2, check where it
is approaching with 100 or more itrations.
import math
x = 0
for n in range(1, 20):
    x = math.sqrt(2 + x)
print(x)

1.9999999999910236
```

Above, We do not knew how much iteration will we need to reach at some value.(we used hit & trial method to decide when to terminate).

- we can do it in better way, if the difference between the previous value and the current value in the sequence is less than some predefined value (tolerance), then we terminate the iteration. because after that we would not get much difference (in the approached values) & we`ll losse our time instead.

```
import math
x_prev, x_curr = 0, math.sqrt(2)
tol, count = 0.00001, 0
while abs(x_curr - x_prev) >= tol:
    x_prev = x_curr
    x_curr = math.sqrt(2 + x_prev)
    count += 1
print(f'Value of x at {tol} tolerance is {x_curr}')
print(f'It took {count} iterations')

Value of x at 1e-05 tolerance is 1.9999976469034038
It took 9 iterations
```

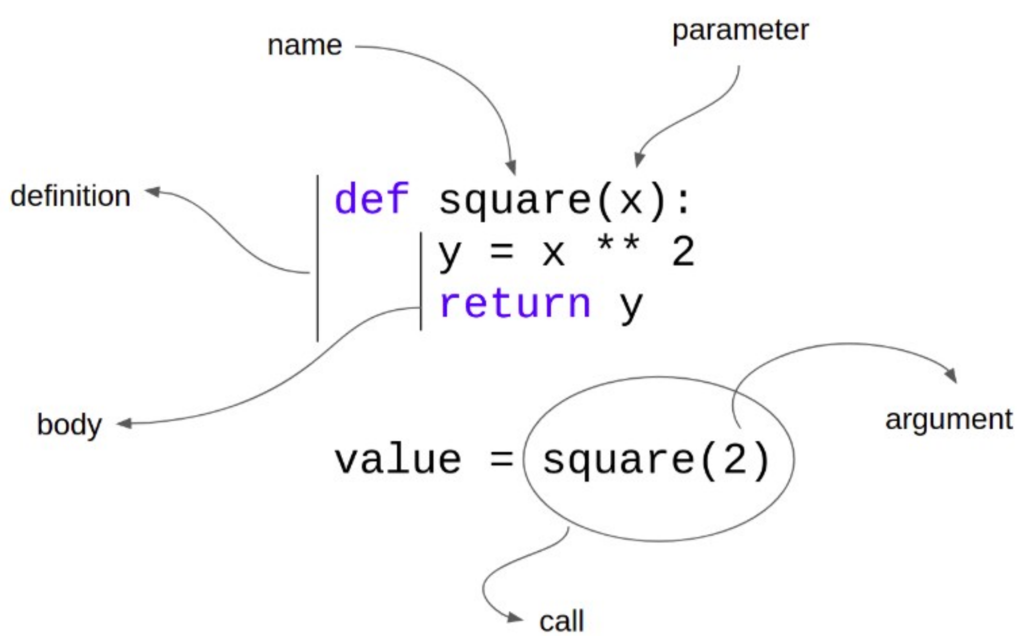
Some Libraries

- Random
- `random.randit(1,n)` this will give any random number b/w 1 & n.

- `random.choice('AmitisAgOODbOy')` this will give any random character of the word AmitisAgOODbOy. It basically takes any sequence & give any random item of the sequence as output.

Functions

- Functions have to be defined before they can be called. The function call cannot come before the definition.
- Function calls could be used in expressions: `if square(x) + square(y) == square(z):`
- Function calls cannot be assigned values: `square(x) = 5`
- Functions can be called from within other functions.
- Functions can be defined inside other functions.



Functions could have multiple return statements, but the moment the first return is executed, control exits from the function:

```
def foo():
    return 1
    return 2

print(foo())

1
```

An example of a function having multiple returns that are not redundant:

```python
def evenOrOdd():
  n = int(input())
  if n % 2 == 0:
      return 'even'
  else:
      return 'odd'

print(evenOrOdd())
```

```
21
odd
```

Docstrings

- A docstring is a string literal that occurs as the first statement in a module, function, class, or method definition. Such a docstring becomes the `__doc__` special attribute of that object.

```python
def square(x):
    """Return the square of x."""
    return x ** 2

print(square.__doc__)
```

```
Return the square of x.
```

Arguments

- Positional arguments
- Arguments are passed to the parameters of the function based on the position they occupy in the function call.
- Positional arguments are also called required arguments. meaning, there should be exactly as many arguments in the function call as there are parameters in the function definition.
- Keyword arguments
- the names of the parameters are explicitly specified and the arguments are assigned to it using the = operator.
- Default arguments

```python
#Function defination

def isRight(x, y, z):
    if x ** 2 + y ** 2 == z ** 2:
        return True
    return False
```

```
#Function call as positional arguments.
isRight(1,2,3)  # 1,2,3 corresponding to x,y,z respectively according
to the position of the parameters of the function(same order).

#Function call as keyword arguments.
isRight(x = 3, y = 4, z = 5)

#Keyword arguments and positional arguments can be combined in a
single call.
isRight(3, y = 4, z = 5)      #right way

#Whenever both positional and keyword arguments are present in a
function call, the keyword arguments must always come at the end.
#isRight(x = 3, 4, 5)          #wrong way : Intreprator will throw
error.


#this will thro error as isRight() got multiple values for argument x.
#isRight(3, x = 3, y = 4, z = 5)

  File "<ipython-input-299-826cfb3470f4>", line 18
    isRight(x = 3, 4, 5)          #wrong way : Intreprator will throw
error.
                       ^
SyntaxError: positional argument follows keyword argument
```

Default argument

- Parameters that are assigned a value in the function definition are called default parameters.
- Default parameters always come at the end of the parameter list in a function definition.
- The argument corresponding to a default parameter is optional in a function call.
- An argument corresponding to a default parameter can be passed as a positional argument or as a keyword argument.

```
#The parameter metric has 'manhattan' as the default value

##def distance(metric = 'manhattan', x, y): It is a wrong way to
define the parameters, This code throws a SyntaxError with the
following message: non-default argument follows default argument.
# In the function definition, the default parameter must always come
at the end of the list of parameters.
def distance(x, y, metric = 'manhattan'):
    if metric == 'manhattan':
        return abs(x) + abs(y)
    elif metric == 'euclidean':
        return pow(x ** 2 + y ** 2, 0.5)
```

```
# all three are equivalent function call according to the above
function.
distance(3, 4)
distance(3, 4, 'manhattan')
distance(3, 4, metric = 'manhattan')

#the two are equivalent.
distance(3, 4, metric = 'euclidean')
distance(3, 4,'euclidean')
```

```
5.0
```

Call by value

- This kind of a function call where the value in a variable is passed as argument to the function is called **call by value**.
- The value of num (5) is passed as an argument to the function.
- Inside the function, x is created as a local variable and initialized with the value 5.
- Then, x = x * 2 → x becomes 10.
- 10 is returned.

```
def double(x):
    x = x * 2
    return x

a = 5
print(f'before function call, a = {a}')
double(a)
print(f'after function call, a = {a}')
```

```
before function call, a = 5
after function call, a = 5
```

Scope : The region in the code where a name can be referenced is called its scope.

- Local vs Global
- **Local:** Whenever a variable is assigned a value anywhere within a function, its scope becomes local to that function. In other words, whenever a variable appears on the **left side** of an assignment statement anywhere within a function, it becomes a local variable.
- **Global:** If a variable is only referenced inside a function and is never assigned a value inside it, it is implicitly treated as a global variable.

Namespaces:

- It is about simply the names assigned to any class, function or variable etc and were the interpreter is storing them.
- globals() : Global names (names assigned in Global)
- locals() : Local names (names assigned in local)

Built-ins

- keywords like dict, int, str etc.

Whenever the interpreter comes across a name in a function it sticks to the following protocol:

- First peep into the local namespace created for that function call to see if the name is present in it. If it is present, then go ahead and use the value that this variable points to in the local namespace.
- If it is not present, then look at the global namespace. If it is present in the global namespace, then use the value corresponding to this name.
- If it is not present in the global namespace, then look into the built-in namespace. We will come back to the built-in namespace right at the end.
- If it is not present in any of these namespaces, then raise a NameError.

```
# x becomes local since it is in left side & not assigned any value
before in the local.

def foo():
  print(x)
  #x = x + 1     # UnboundLocalError: cannot access local variable 'x'
where it is not associated with a value

x = 10
foo()

10

# To make it correct we have to say to the interpreter that we are
using global variable here in the left side.
def foo():
  global x
  print(x)
  x = x + 1

x = 10
foo()

10
```

List

- A list in Python is a data structure that is used to store a sequence of objects(of different type also).
- we can slice & indexed, find length using `len` same as string. Also, make any sequence using `list()` keyword. example, `list(range(10))` resulting `[0,1,2,3,4,5,6,7,8,9]`.
- List is itrable.
- we can use **list concatenation** using `+` operator & `list.append(item_to_be added)` keyword to grow it.

```
numbers = [1, 2, 3]
print(type(numbers))
print(isinstance(numbers, list))
```

```
<class 'list'>
True
```

```
list1 = []          # [1]
list2 = list()    # [2]

list1 = list1 + [1]
print(list1)
list2.append(2)
print(list2)


list3 = list1 + list2
list4 = list2 + list1
print(list3)     #[1,2]
print(list4)     #[2,1]


print(list2 == list3)      # compares the list in Lexicographic ordering.
print(list2 > list3)

print(max(list3))
print(sum(list3))

print(sorted([1,4,7,2,3]))

list6 = [3,9,1,6,8]
print(list6.sort())    #.sort do not return anything it just sort the list permanently.
list6.sort()
print(list6)
```

```
[1]
[2]
[1, 2]
```

```
[2, 1]
False
True
2
3
[1, 2, 3, 4, 7]
None
[1, 3, 6, 8, 9]
```

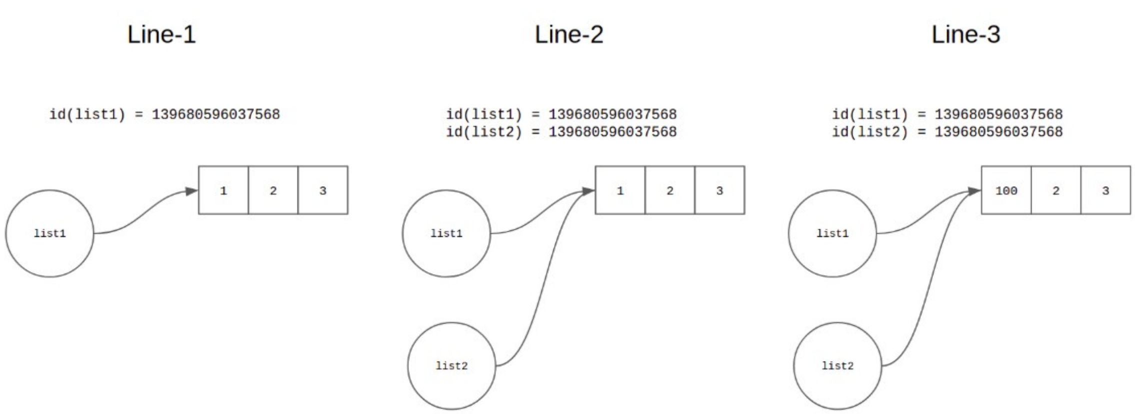
```python
list5 = list()
list5 = list5.append(3)   # he .append() method modifies the list in place and returns None. so list5 = None
list5.append(3)       # this will give error because of None.apend(3) instead of list5.append(3).
print(list5)
```

```
[3]
```

Some properties of list.

- lists are mutable. Meaning, we can change the element of the list.
- `list[3] = 5` : Change the 4th(3rd indexed) elememnt of the list to 5.
- Every object in Python has a unique identity: if x is an object, then id(x) returns this object's identity.
- In line no. 2, we are not creating a new object. We are merely creating another name, also called an alias, for the same object.

```python
1   list1 = [1, 2, 3]
2   list2 = list1
3   list2[0] = 100
```



```
list1 = [1, 2, 3]
list2 = list1
list2[0] = 100
print(list1)
print(list2)

print(list1 is list2)   # It means list1 & list2 has the same identity, meaning he`s a single person but he have two names as list1 & list.
```

```
[100, 2, 3]
[100, 2, 3]
True
```

What happens here ?

- list1 and list2 point to two different objects and consequently have different identities. But, they store the same sequence of items and are hence equal.

```
list1 = [1, 2, 3]
list2 = [1, 2, 3]
print(list1 == list2)
print(list1 is list2)
```

```
True
False
```

We can create different object of an object, so that change in one will not affact the other.

```
list1 = [1, 2,  3]
list2 = list(list1)
list3 = list1[:]
list4 = list1.copy()

list2[0] = 100
list3[0] = 200
list4[0] = 300

print(list1, list2, list3, list4)
print(list1 is not list2, list1 is not list3, list1 is not list4)
```

```
[1, 2, 3] [100, 2, 3] [200, 2, 3] [300, 2, 3]
True True True
```

Call by reference

```
# Snippet-1
def foo():
    L.append(1)

L = [0]
print(f'L before: {L}')
foo()
print(f'L after: {L}')
```

`Snippet-1` doesn't have any parameters. Since `L` is not being assigned a new value inside `foo`, the scope of `L` remains global.

```
1 # Snippet-2
2 def foo(L_foo):
3     L_foo.append(1)
4     print(L is L_foo)
5
6 L = [0]
7 print(f'L before: {L}')
8 foo(L)
9 print(f'L after: {L}')
```

`Snippet-2` has `L_foo` as a parameter whose scope is local to `foo`. But note that modifying `L_foo` within the function changes `L` outside the function. This is because, `L_foo` and `L` point to the same object. How did this aliasing happen? The function call at line-8 works something like an assignment statement: `L_foo = L`, so `L_foo` is just another name that refers to the object that `L` is bound to. This type of function call where a reference to an object is passed is termed call by reference. Whenever a mutable variable is passed as an argument to a function, the references to the corresponding object are passed.

If all this seems too complicated, just remember that modifying mutable objects within a function produces side effects outside the function. What if we don't want these side effects? We have to create a new list object like we did before:

```
1 def foo(L_foo):
2     L_foo.append(1)
3     print(L is L_foo)
4
5 L = [0]
6 print(f'L before: {L}')
7 foo(list(L))
8 print(f'L after: {L}')
```

`foo` doesn't produce any side effects. Line-7 could be replaced with `foo(L[:])` or `foo(L.copy())`.

```
# Snippet-2
def foo(L_foo):
    L_foo.append(1)
    print(L is L_foo)     # True because L & L_foo is pointing to the
same object despite of being one local & other global variable.

L = [0]
print(f'L before: {L}')
foo(L)
print(f'L after: {L}')
```

```
L before: [0]
True
L after: [0, 1]
```

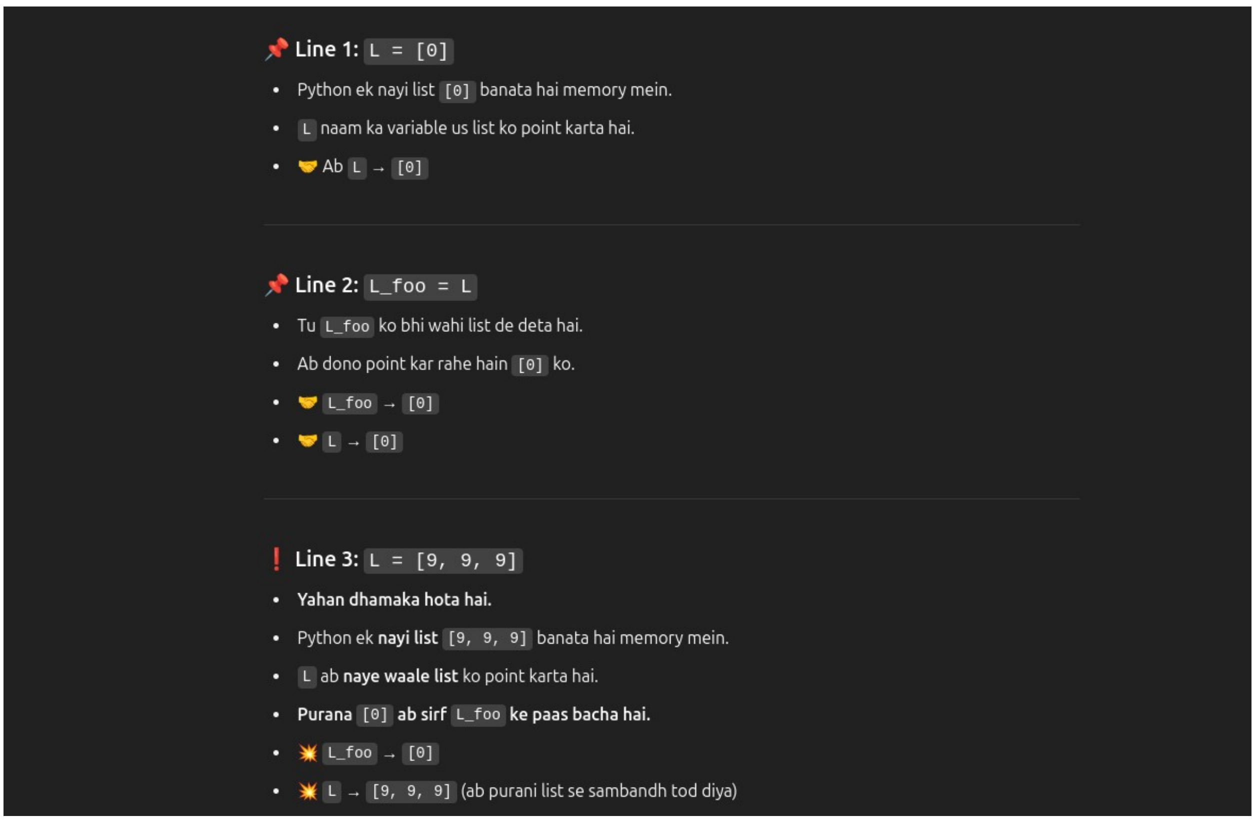
```
def foo(L_foo):
    L_foo.append(1)
    print(L is L_foo)

L = [0]
```

```python
print(f'L before: {L}')
foo(list(L))
print(f'L after: {L}')
```

```
L before: [0]
False
L after: [0]
```

Here, L is a global variable residing in global namespace but L_foo in 2nd snippet is a logal variable reciding in local namespace but pointing to the same object L in global namespace till the time L is not reassigned.

when L is reassigned (let, [9, 9, 9]), but since now L is reassigned to some other list, L will point to that list. and L_foo will point to the same list that L was pointing before the reassigment (let [0]).



📌 **Line 1:** `L = [0]`

- Python ek nayi list `[0]` banata hai memory mein.
- `L` naam ka variable us list ko point karta hai.
- 🤝 Ab `L` → `[0]`

📌 **Line 2:** `L_foo = L`

- Tu `L_foo` ko bhi wahi list de deta hai.
- Ab dono point kar rahe hain `[0]` ko.
- 🤝 `L_foo` → `[0]`
- 🤝 `L` → `[0]`

❗ **Line 3:** `L = [9, 9, 9]`

- **Yahan dhamaka hota hai.**
- Python ek **nayi list** `[9, 9, 9]` banata hai memory mein.
- `L` ab **naye waale list** ko point karta hai.
- **Purana `[0]` ab sirf `L_foo` ke paas bacha hai.**
- 💥 `L_foo` → `[0]`
- 💥 `L` → `[9, 9, 9]` (ab purani list se sambandh tod diya)

`list(L)` list of list nahi banata, balki ek naya list banata hai jo wahi elements copy karta hai jo L ke andar hain.

□ To kab `[[0]]` banta hai?

`list([[0]])` ya `L = [[0]]`

```python
L = [0]
L_foo = L
L = [9, 9, 9]    # ab L naye list ko point kar raha hai
```

```
print(L_foo)    # still [0]
print(L)
```

```
[0]
[9, 9, 9]
```

choices is a function in the **random library**. It uniformly samples from the seven numbers (0 to 6) given in the input list with replacement.

```
import random
runs = random.choices([0, 1, 2, 3, 4, 5, 6], k = 120)    # choosing 
any random number from range(0,7) with replacement & storing it in a 
list named runs for 120 times.
print(type(runs))
print(len(runs))
```

```
<class 'list'>
120
```

```
for run in [0, 1, 2, 3, 4, 5, 6]:
    print('{} appears {} times'.format(run, runs.count(run)))
```

```
0 appears 13 times
1 appears 14 times
2 appears 15 times
3 appears 17 times
4 appears 19 times
5 appears 23 times
6 appears 19 times
```

The counts are quite close. But this is not very practical:

- 5 runs are seldom observed in cricket matches.
- 0, 1 and 2 are much more common than 3, 4 and 6.

We can give our preferences using a weights keyword-argument:

```
# weights => "relative probability" : This outcome is twice as likely 
as that
# one. we are not giving absolute probabilities, just comparing 
likelihoods
# relative to each other. %%

import random
# choices is distributed over multiple lines
# this is done to improve readability

#weights => "relative probability" :  This outcome is twice as likely 
as that one. we are not giving absolute probabilities, just comparing 
likelihoods relative to each other.
```

```
runs = random.choices([0, 1, 2, 3, 4, 5, 6],
                      weights = [30, 30, 20, 4.8, 10, 0.2, 5],
                      k = 120)
for run in [0, 1, 2, 3, 4, 5, 6]:
    print('{} appears {} times'.format(run, runs.count(run)))
print(f'Total number of runs scored = {sum(runs)}')

print(runs)
```

```
0 appears 30 times
1 appears 38 times
2 appears 31 times
3 appears 5 times
4 appears 9 times
5 appears 0 times
6 appears 7 times
Total number of runs scored = 193
[0, 2, 1, 1, 2, 0, 0, 3, 2, 3, 1, 0, 3, 1, 4, 1, 0, 0, 2, 2, 1, 0, 0,
2, 0, 1, 2, 0, 1, 1, 2, 1, 3, 1, 0, 1, 3, 6, 1, 2, 2, 0, 0, 2, 1, 6,
2, 0, 1, 1, 1, 6, 1, 2, 2, 0, 4, 1, 6, 6, 1, 0, 0, 2, 1, 2, 4, 2, 0,
4, 2, 2, 1, 0, 2, 4, 0, 1, 1, 1, 2, 2, 4, 1, 0, 0, 1, 1, 2, 2, 0, 1,
0, 1, 2, 1, 1, 1, 0, 4, 1, 1, 4, 0, 1, 0, 2, 2, 2, 1, 4, 2, 0, 2, 6,
1, 0, 6, 2, 0]
```

index is a method that accepts an element as input and returns the first occurrence of this element in the list.

```
first_six_ball = runs.index(6) + 1    # .index starts counting from 0
(index of elements of list) that is why adding here 1 to get the exact
position.
print(first_six_ball)
```

```
38
```

```
# 5 never occurs in the list. so It`ll throw an ValueError. (try using
try & except here to handle excepions)

#first_five_ball = runs.index(5)          #uncomment this line & run
#print(first_five_ball)
```

enumerate : enumerate(iterable, start) output: index & element of any itrable`s element..

- default : start = 0

List

Tuple

String

Set ▯ (order unpredictable hota hai)

Dictionary ▯ (lekin sirf keys ya items ko loop karte waqt)

Generators / Iterators ▯

```python
#list
for idx, val in enumerate(['a', 'b', 'c']):
    print(idx, val)
```

```
0 a
1 b
2 c
```

```python
#string
for idx, char in enumerate("hello"):
    print(idx, char)
```

```
0 h
1 e
2 l
3 l
4 o
```

```python
#tuple
for idx, item in enumerate((10, 20, 30)):
    print(idx, item)
```

```
0 10
1 20
2 30
```

```python
#Dictionary(keys)
d = {'a': 1, 'b': 2}
for idx, key in enumerate(d):
    print(idx, key)
```

```
0 a
1 b
```

`.items()` is a property of a dictionary which gives both pair `(key, value)` of each element.

▯ Ek iterable object hai, jisme tuples hote hain.

(Technically: `dict_items` type ka object hota hai jo iterable hai.)

```python
for i in d.items():
  print(i, type(i))
```

```
('a', 1) <class 'tuple'>
('b', 2) <class 'tuple'>
```

```python
#Dictionary(items)
for idx, (key, value) in enumerate(d.items()):
    print(idx, key, value)
print(d.items())
```

```
0 a 1
1 b 2
dict_items([('a', 1), ('b', 2)])
```

```python
#Generator
gen = (x*x for x in range(3, 6))
for idx, val in enumerate(gen):
    print(idx, val)
```

```
0 9
1 16
2 25
```

```python
for ball, run in enumerate(runs):
    if run == 6:
        print(f'The first six was hit at ball number {ball + 1}')
        break
```

```
The first six was hit at ball number 38
```

```python
# find the number of balls it took to score the last 50 runs in the
innings?

ball = 0
last_run = 0

for run in reversed(runs):              #The reversed object helps us
iterate through the list in the reversed order.
  last_run += run
  ball += 1

  if last_run >= 50:
    print(f'we scored {last_run} runs in {ball} balls!')
    break
```

```
we scored 50 runs in 31 balls!
```

List Methods

- insert : `.insert(index, value)`
- value with index out of range will insert in the end of the list.
- pop : `.pop(index)` , default: index = 0
- reverse : `.reverse()`

- this method replaces the new reversed list with the previous original list & not return anything.
- sort : `.sort()` and `.sort(reverse = True)`
- `reverse = True` will reverse the list, and not return anything.
- remove : `.remove(value)`
- It`ll remove the first value it finds in the list (itrating from the left to right)

 Stack (Last In First Out - LIFO)

- A stack is a linear data structure that follows the LIFO (Last In, First Out) principle.

 Queue (First In First Out - FIFO)

- A queue is a linear data structure that follows the FIFO (First In, First Out) principle.

Strings and Lists

- split : `string.split(separator, limit)`
- split is a string method that splits a string along a delimiter.
- join : `string.join(iterable)`
- join the itrable with the provided string.

```
# insert

L = [10, 20, 30]
L.insert(0, 5)          # L becomes [5, 10, 20, 30]
L.insert(2, 15)         # L becomes [5, 10, 15, 20, 30]
L.insert(4, 25)         # L becomes [5, 10, 15, 20, 25, 30]
L.insert(len(L), 35)    # L becomes [5, 10, 15, 20, 25, 30, 35]
L.insert(20, 40)        # L becomes [5, 10, 15, 20, 25, 30, 35, 40]

#pop

L = ['a', 'b', 'c', 'd', 'e', 'f']
index = 1
x = L.pop(index)
print(f'The element {x} at index {index} was removed from the list.')
print(f'The current list is {L}')

The element b at index 1 was removed from the list.
The current list is ['a', 'c', 'd', 'e', 'f']

# reverse

L = [1, 2, 3, 4, 5]
print('Before:', L, id(L))
```

```
L.reverse()
print('After:', L, id(L))
```

```
Before: [1, 2, 3, 4, 5] 136787245619840
After: [5, 4, 3, 2, 1] 136787245619840
```

```
L = [1, 2, 3, 4, 5]
L = L.reverse()    # here, it replaces the new reversed list(5, 4, 3, 2, 1]) with the previous original list(1, 2, 3, 4, 5]) & not return anything.
print(L)
```

```
None
```

```
# stack  LIFO

stack_list = []

stack_list.append('Harry Potter and the Chamber of Secrets')
stack_list.append('chapter 2')

print(stack_list)

stack_list.pop()

print(stack_list)
```

```
['Harry Potter and the Chamber of Secrets', 'chapter 2']
['Harry Potter and the Chamber of Secrets']
```

```
# queue FIFO

queue_list = []

queue_list.insert(0, 'Harry Potter and the Chamber of Secrets')
queue_list.insert(0, 'chapter 2')

print(queue_list)

queue_list.pop()

print(queue_list)
```

```
['chapter 2', 'Harry Potter and the Chamber of Secrets']
['chapter 2']
```

```
words = ['this', 'sentence', 'is', 'false']
sentence = words[0]

print(type(sentence))
```

```
<class 'str'>
```

assert

- `assert condition`
- If the conditional expression following the assert keyword is True, then control transfers to the next line. If it is False, the interpreter raises an AssertionError.

```python
import random

runs = random.choices([0,1,2,3,4,5,6], weights = [15, 25, 25, 10, 15, 0.4, 9.6], k = 120)

assert len(runs) == 120

overs = []
new_over = []

for ball, run in enumerate(runs):
  new_over.append(run)

  if (ball + 1) % 6 == 0:
    overs.append(new_over)
    new_over = []

assert len(overs) == 20
for ball in overs:
  assert len(ball) == 6
```

Matrices

- 2D matrix
- Shallow and Deep Copy

```python
# 2D matrices

import random

mat = []
for i in range(3):
  row = []
  for j in range(3):
    row.append(random.randint(1, 100))
  mat.append(row)

print(mat)
```

```
[[13, 70, 99], [2, 20, 60], [48, 87, 61]]
```

```python
# Here, change in one matrix is changing others too :
#Lists are mutable. mat2 is just an alias for mat1 and both point to the same object. Modifying any one of them will modify both.
```

```
# mat1 = [[46, 12, 30], [25, 84, 54], [78, 88, 97]]
# mat2 = mat1
# mat1[0][0] = 100

# print(mat1)
# print(mat2)
```

`mat2 = mat1.copy()` creates a shallow copy.

- We have a mutable object inside another mutable object. In such a case copy just does a shallow copy; only a new outer-list object is produced. This means that the inner lists in mat1 and mat2 are still the same objects:

```
mat1 = [[46, 12, 30], [25, 84, 54], [78, 88, 97]]
mat2 = mat1.copy()

mat2[0][0] = 100

print(mat1)
print(mat2)

[[100, 12, 30], [25, 84, 54], [78, 88, 97]]
[[100, 12, 30], [25, 84, 54], [78, 88, 97]]
```

to solve this problem, we should use deepcopy

```
from copy import deepcopy
mat1 = [[1, 2], [3, 4]]
mat2 = deepcopy(mat1)
print(mat1 is not mat2)
print(mat1[0] is not mat2[0])
print(mat1[1] is not mat2[1])

True
True
True
```

Tuples

- Introduction
- It is a immutable sequence of values. but, the element inside it can be mutable. for example, `([1,2], "amit", [3,6,7])` this tuple is immutable but inside it the lists are mutable.
- They can be indexed and sliced just like lists.
- We can iterate through a tuple.
- **count and index** are the **only two methods** which are defined for tuple.

- More on Tuples
- A list can be converted into a tuple and vice versa.
- `a_tuple = (1, 'cool', True)` : It can hold a non-homogeneous sequence of items
- tuple of tuples & tuple of list is possible.
- Lists and Tuples
- visit the following image to find difference between the two.
- Packing and Unpacking

```
# Introduction
numbers = (1, 2, 3, 1, 1)
print(numbers.count(1))
print(numbers.index(2))
```

```
3
1
```

```
#singleton tuple
i_am_single = (1, )
print(len(i_am_single))
print(isinstance(i_am_single, tuple))

i_am_not_a_tuple = (1)
print(isinstance(i_am_not_a_tuple, int))
```

```
1
True
True
```

a tuple is immutable, the element inside it is mutable

```
a_tuple = ([0, 1, 2], 3, 4, "amit")
a_tuple[0][0] = 100

a_tuple
```

```
([100, 1, 2], 3, 4, 'amit')
```

```
a_tuple = ([0, 1, 2], [4, 5, 6])
print(id(a_tuple[0]))
a_tuple[0][0] = 100
print(id(a_tuple[0]))

# a_tuple[0] = 'Amit'
# a_tuple[0] = [1,2,3]
```

```
136787245295040
136787245295040
```

Lists and Tuples

We have seen the close kinship between lists and tuples. Here is a brief summary that highlights the points of agreement and disagreement:

List	Tuple
Mutable	Immutable
`L = [1, 2, 3]`	`T = (1, 2, 3)`
Supports indexing and slicing	Supports indexing and slicing
Supports item assignment	Doesn't support item assignment
Supported methods: `count, index, append, insert, remove, pop` and others	Supported methods: `count, index`
To get a list: `list(obj)`	To get a tuple: `tuple(obj)`

The partnership between lists and tuples is quite interesting and can be explored further with another example.

> Populate a list that contains all ordered pairs of positive integers whose product is 100. Note that order matters: (2, 50) and (50, 2) are two different pairs.

```
populated_list = []

for i in range(1,101):
  for j in range(1,101):
    if i*j == 100:
      populated_list.append((i,j))
print(populated_list)
```

```
[(1, 100), (2, 50), (4, 25), (5, 20), (10, 10), (20, 5), (25, 4), (50, 2), (100, 1)]
```

Tuple packing. **only for tuple**

```
T = 1, 2, 3       #assigning in this way makes the variable to point towards a tuple.
print(T)
print(isinstance(T, tuple))
```

```
(1, 2, 3)
True
```

```
def max_min(a, b):
    if a > b:
        return a, b
    return b, a

x = max_min(1, 2)
print(x)
print(isinstance(x, tuple))
```

```
(2, 1)
True
```

The reverse operation is called sequence unpacking:

```
x, y, z = T
print(x, y, z)
```

```
1 2 3
```

It works for every data structure in python. **Unpacking only**

```
l1, l2, l3, l4 = 'good'         # string
num1, num2, num3 = [1, 2, 3]    # list
b1, b2 = (True, False)          # tuple
x, y, z = range(3)              # range
```

hashing

- An object is hashable if it has a fixed hash value that does not change during its lifetime and it can be compared to other objects using ==.
- In Python terms, for an object to be a key of a dictionary:
- The object must have a **hash**() method.
  - Ye method object ko ek unique number (hash value) de deta hai, jo use dictionary ke bucket me rakhne ke liye use hota hai.
- It must be immutable, or at least act like it's immutable.
  - Object ka data badla nahi ja sakta after creation.
- It must also have a working **eq**() method.
  - Ye method check karta hai ki do objects barabar hain ya nahi (equal hain ya nahi).
  - Hash collision ho sakta hai (do keys ka hash same ho jaye). Us case me Python check karta hai: "Kya yeh dono keys sach me same hain?" → using **eq**().

Dictionaries

- Introduction
- A dictionary is a collection of key-value pairs.
- To add new pair or update into : `dict[key] = value`
- To delete a pair: `dict[key].pop()`
- we can use `keys` as any data stucture which are immutable, hashable & comparable.
- example, a tuple of lists is immutable but is not hashable(because list is not hashable) so can not be used as a key.

- **view object** of dictionary : live chnage, means chnaging in dictionary will chnage the values of these object.
- keys = dict.keys()
- values = dict.values()
- items = dict.items()

Note : immutable means we can not change the element of the object. (but can be itrable)

```
# dict1 & dict2 are different because dict_1 does not contain any
mutable object.
dict_1 = {'one': 1, 'two': 2, 'three': 3}
dict_2 = dict_1.copy()      # dict(dict_1) also works
dict_2['four'] = 4
print(dict_1, dict_2)
print(dict_1 is not dict_2)

{'one': 1, 'two': 2, 'three': 3} {'one': 1, 'two': 2, 'three': 3,
'four': 4}
True

#here, we need to use deepcopy because dict1(mutable object:
dictionary) contains another mutable object.
from copy import deepcopy
dict_1 = {'one': [1], 'two': [1, 1], 'three': [1, 1, 1]}
dict_2 = deepcopy(dict_1)
dict_2['one'].append(100)
print(dict_1, dict_2)
print(dict_1 is not dict_2)
print(dict_1['one'] is not dict_2['one'])

{'one': [1], 'two': [1, 1], 'three': [1, 1, 1]} {'one': [1, 100],
'two': [1, 1], 'three': [1, 1, 1]}
True
True
```

Some data analysis using dictionary

```
text = "In reality, programming languages are how programmers express
and communicate ideas — and the audience for those ideas is other
programmers, not computers. The reason: the computer can take care of
itself, but programmers are always working with other programmers, and
poorly communicated ideas can cause expensive flops. In fact, ideas
expressed in a programming language also often reach the end users of
the program — people who will never read or even know about the
program, but who nevertheless are affected by it."
```

```
sentences = text.split('.')
sentences.remove('')
assert len(sentences) == 3

filtered_words = []
for sent in sentences:
  unfiltered_word = sent.split(' ')
  for words in unfiltered_word:
    words = words.lower()
    if not(words == '' or words == '–'):
      if not words.isalnum():
        words = words[:-1]
        filtered_words.append(words)
      else:
        filtered_words.append(words)

len(filtered_words)
```

```
82
```

```
uniq_words = dict()
for word in filtered_words:
    if word not in uniq_words:
        uniq_words[word] = 0
    uniq_words[word] += 1
print(f'There are {len(uniq_words)} unique words in this text')
print(uniq_words)
```

```
There are 58 unique words in this text
{'in': 3, 'reality': 1, 'programming': 2, 'languages': 1, 'are': 3,
'how': 1, 'programmers': 4, 'express': 1, 'and': 3, 'communicate': 1,
'ideas': 4, 'the': 6, 'audience': 1, 'for': 1, 'those': 1, 'is': 1,
'other': 2, 'not': 1, 'computers': 1, 'reason': 1, 'computer': 1,
'can': 2, 'take': 1, 'care': 1, 'of': 2, 'itself': 1, 'but': 2,
'always': 1, 'working': 1, 'with': 1, 'poorly': 1, 'communicated': 1,
'cause': 1, 'expensive': 1, 'flops': 1, 'fact': 1, 'expressed': 1,
'a': 1, 'language': 1, 'also': 1, 'often': 1, 'reach': 1, 'end': 1,
'users': 1, 'program': 2, 'people': 1, 'who': 2, 'will': 1, 'never':
1, 'read': 1, 'or': 1, 'even': 1, 'know': 1, 'about': 1,
'nevertheless': 1, 'affected': 1, 'by': 1, 'it': 1}
```

```
print(uniq_words['programmers'])
```

```
4
```

```
for words in filtered_words:
  if "programmer" in words:
    print(words)
```

```
programmers
programmers
```

```
programmers
programmers
```

```python
word1 = word2 = word3 = ''
val1 = val2 = val3 = 0

data = {}
for word,freq in uniq_words.items():
  if freq > val1 :
    val1,val2,val3 = freq, val1, val2
    word1, word2, word3 = word, word1, word2
  elif freq > val2 and freq < val1 :
    val2, val3 = freq, val2
    word2, word3 = word, word2
  elif freq > val3 and freq < val2:
    val3 = freq
    word3 = word

data[word1] = val1
data[word2] = val2
data[word3] = val3

print(data)
```

```
{'the': 6, 'programmers': 4, 'in': 3}
```

```python
pangram = 'the quick brown fox jumps over the lazy dog'

words = pangram.split(' ')
splitted_letters = ''.join(words)
alphabets = sorted(splitted_letters)

mapping, count = dict(), 0

for count, letter in enumerate(alphabets):
  if letter not in mapping :
    mapping[letter] = count +1


print(mapping)
```

```
{'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5, 'f': 8, 'g': 9, 'h': 10, 'i':
12, 'j': 13, 'k': 14, 'l': 15, 'm': 16, 'n': 17, 'o': 18, 'p': 22,
'q': 23, 'r': 24, 's': 26, 't': 27, 'u': 29, 'v': 31, 'w': 32, 'x':
33, 'y': 34, 'z': 35}
```

view object

```python
keys = mapping.keys()
values = mapping.values()
items = mapping.items()
```

```
print(keys)
print(values)
print(items)



print('a' in mapping.keys())
print(1 in mapping.values())
print(('a', 1) in mapping.items())


mapping['ab'] = 3          # some noise added to mapping
value = mapping.pop('ab')
print(value)
print('ab' not in mapping)
```

```
dict_keys(['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'])
dict_values([1, 2, 3, 4, 5, 8, 9, 10, 12, 13, 14, 15, 16, 17, 18, 22,
23, 24, 26, 27, 29, 31, 32, 33, 34, 35])
dict_items([('a', 1), ('b', 2), ('c', 3), ('d', 4), ('e', 5), ('f',
8), ('g', 9), ('h', 10), ('i', 12), ('j', 13), ('k', 14), ('l', 15),
('m', 16), ('n', 17), ('o', 18), ('p', 22), ('q', 23), ('r', 24),
('s', 26), ('t', 27), ('u', 29), ('v', 31), ('w', 32), ('x', 33),
('y', 34), ('z', 35)])
True
True
True
3
True
```

```
key_list = list(mapping.keys())
print(keys)
print(key_list)
```

```
dict_keys(['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']
```

Sets : unordered collection of unique, immutable elements.

- **Used to** : membership testing, eliminating duplicate entries, and performing mathematical set operations like union, intersection, and difference.

- Any hashable object can be added to sets.

- `set.add(element_to_be_added)` , set method used to add any element is `.add` .

- `set.remove('element_to_be_removed')` , we can use set method `.remove` to remove any element from the set.
- list is mutable means not hashable, hence can not added to a set.
- tuple of list is not hashable, hence can not added to a set.
- set is itrable.
- **Key Characteristics of Sets:**
- **Unordered:** The elements in a set do not maintain any specific order.
- **Unique Elements:** A set automatically removes duplicate values; each element is unique.
- **Mutable:** While the elements themselves must be immutable (e.g., numbers, strings, tuples), the set as a whole is mutable, allowing you to add or remove elements after its creation.
- **Unindexed:** Sets do not support indexing, slicing, or other sequence-like behavior.
- Both dictionary and set use `{}` as symbols, but:
  - If you write `{}` without anything, it creates an empty dictionary, not a set.
  - To create an empty set, use `set()`.

```
# set automatically removes the duplicates.
nums_1 = {2, 4, 6, 8, 10}
nums_2 = {2, 2, 4, 4, 6, 6, 8, 8, 10, 10}
print(nums_1, nums_2)
print(nums_1 == nums_2)
print(nums_1 is not nums_2)
```

```
{2, 4, 6, 8, 10} {2, 4, 6, 8, 10}
True
True
```

```
num = 1
digits = set()
for i in range(100):
    num *= 7
    last = num % 10
    digits.add(last)
print(digits)
```

```
{9, 3, 1, 7}
```

```python
# Consider the first 100 powers of 7:
# Note down the last digit of each of these powers. How many of them are unique? What are these numbers?

num = 1
digits = set()
for i in range(100):
  num *= 7
  last = num % 10
  digits.add(last)
print(digits)
```

```
{9, 3, 1, 7}
```

Set operations

- **Subset (⊆)**: Set A is a subset of set B if every element of A is also in B. ::::::::: `A <= B` or `A.issubset(B)`
- **Proper Subset (⊂)**: Set A is a proper subset of B if A ⊆ B and A ≠ B. :::::::::: `A < B`
- **Superset (⊇)**: Set A is a superset of B if every element of B is in A. ::::::::::: `A.issuperset(B)` or `B >= A`
- **Proper Superset (⊃)**: Set A is a proper superset of B if A ⊇ B and A ≠ B. ::::::::::: `A > B`
- **Union (∪)**: The union of sets A and B is the set of elements that are in A, in B, or in both. ::::::::::: `A.union(B)` or `A | B`
- **Itersection (∩)**: The intersection of sets A and B is the set of elements common to both A and B. ::::::::::: `A.intersection(B)` or `A & B`
- Difference (−): The difference of sets A and B is the set of elements in A that are not in B. ::::::::::: `A.difference(B)` or `A - B`
- **Symmetric Difference (Δ)**: The symmetric difference of sets A and B is the set of elements in either A or B but not in both. ::::::::: `A.union(B) - A.intresection(B)`

```python
#subset
A = {1, 3, 5}
B = {1, 2, 3, 4, 5}
print(A.issubset(B))     # method-1
print(A <= B)            # method-2
```

```
True
True
```

```python
#Propersubset
A = {1, 3, 5}
B = {1, 2, 3, 4, 5}
print(A < B)
```

```
True
```

```python
#Superset
A = {1, 3, 5}
B = {1, 2, 3, 4, 5}
print(B.issuperset(A))
print(B >=A)
```

```
True
True
```

```python
#propersuperset
A = {1, 3, 5}
B = {1, 2, 3, 4, 5}
print(B > A)
```

```
True
```

```python
#Union
A = {1, 3, 5}
B = {1, 2, 3, 4, 5}
print(A.union(B))
print(A | B)
```

```
{1, 2, 3, 4, 5}
{1, 2, 3, 4, 5}
```

```python
#Intersection
A = {1, 3, 5}
B = {1, 2, 3, 4, 5}
print(A.intersection(B))
print(A & B)
```

```
{1, 3, 5}
{1, 3, 5}
```

```python
#Difference
A = {1, 3, 5}
B = {1, 2, 3, 4, 5}
print(B.difference(A))
print(B - A)
print(A -B)
```

```
{2, 4}
{2, 4}
set()
```

```
A = {'this', 'is', 'a', 'set'}
print('Before', A)
A.remove('this')
print('After', A)
```

```
Before {'set', 'a', 'is', 'this'}
After {'set', 'a', 'is'}
```

```
# we know set is mutable but not access by indexing instead we can add
& remove element but not replace directly.

A = {1, 2, 3}
B = A
B.add(4)
print(A, B)
print(A is B)
```

```
{1, 2, 3, 4} {1, 2, 3, 4}
True
```

```
# set is a unorderd & mutable object.
A = {1, 2, 3}
B1 = A.copy()
B2 = set(A)
B1.add(4)
B2.add(0)
print(A, B1, B2)
print(A is not B1)
print(A is not B2)
```

```
{1, 2, 3} {1, 2, 3, 4} {0, 1, 2, 3}
True
True
```

Sets of Immutable Containers with Mutable Contents:

If your set contains immutable containers (like tuples) that reference mutable objects (like lists), then a shallow copy won't suffice. Changes to the mutable objects will reflect in both the original and the copied set.

```
import copy

mutable_list = [1, 2]
original_set = {(1, tuple(mutable_list))}
deep_copied_set = copy.deepcopy(original_set)

deep_copied_set.add("Amit")
print(original_set)
print(deep_copied_set)
```

```
{(1, (1, 2))}
{(1, (1, 2)), 'Amit'}
```

File handling

- **open** : open function returns a file object(itrable line wise line).

📋 Some useful attributes/methods of file object `f`:

Attribute / Method	What it does
`f.read()`	Reads entire file content as a string
`f.readline()`	Reads one line at a time
`f.readlines()`	Reads all lines and returns a list
`f.write("text")`	Writes string to file (write mode)
`f.writelines([...])`	Writes list of strings to file
`f.seek(pos)`	Moves the pointer to given byte position
`f.tell()`	Returns current position of file pointer
`f.close()`	Closes the file

```
f = open('text.txt', 'w')
f.write('Amit\nKumar\nPandey')
f.close()
```

Understanding the Code:

- for line in f: This reads line-by-line from the file.
- After this loop finishes, the file pointer is at the end of the file.
- print(f.read())
- This tries to read what remains in the file.

But since the file pointer is already at the end, it reads nothing, so it prints a blank line.

```
#read : # read is a method which returns a string contanging whole
content of the file with a newline carecter(\n) at the end of each
line excluding last line.
```

```
f = open('text.txt', 'r')
l = []
for i in f.read():   #f.read() is a string of whole content.
  l.append(i)
print(l)
f.close()

f = open('text.txt', 'r')
for line in f :           # f is a file object here.

# first line: "Amit\n", 2nd line: "Kumar\n" & the 3rd line : "Pandey\
n",
#Note: print function print the argument passed to it & move to the
next line.
# so here, for "Amit\n" Amit is printed & move to the new line & then
nature of print moves the cursor to the new line.
# same heppends for other lines too, that is why we can see empty
lines in between the lines.
# try print(line.strip()) : you won`t get any empty lines
#because there is no new line character with the lines now(.strip()
removes leading and trailing whitespace characters from a string),
#so print function is printing each line & move to the next line.

  print(line)
print(f.read())
f.close()
```

```
['A', 'm', 'i', 't', '\n', 'K', 'u', 'm', 'a', 'r', '\n', 'P', 'a',
'n', 'd', 'e', 'y']
Amit

Kumar

Pandey
```

```
f = open('text.txt', 'r')

print(f.read())  # Reads the ENTIRE file and moves the pointer to the
end

for line in f:   # At this point, nothing is left to read
    print(line.strip())

f.close()
```

```
Amit
Kumar
Pandey
```

```python
#readlines : reads one line at a time.
f = open('text.txt', 'r')
line = f.readline()    #reads the entire line including any whitspaces also if any(except the last line).
while line :
  print(line)
  line = f.readline()
f.close()
```

```
Amit

Kumar

Pandey
```

```python
#readlines : reads one line at a time.
f = open('text.txt', 'r')
print(f.readlines())    #reads as a list of all lines(string) of entire content.
f.close()
```

```
['Amit\n', 'Kumar\n', 'Pandey']
```

```python
f = open('writelines.txt', 'w')

f.writelines(["writelines\n", "writes\n", "a\n", 'list\n', 'of\n', 'string\n'])

f.close()
```

```python
f = open('text.txt', 'r')
for line in f :
  print(line)

f.seek(0)
print(f.tell())

print(f.read())
print(f.tell())

f.close()
```

```
Amit

Kumar

Pandey
0
Amit
Kumar
```

```
Pandey
17
```

```
level2 = open('level2.txt', 'r')
print(level2.read())
level2.close()
```

```
Name,Physics,Mathematics,Chemistry
Newton,100,98,90
Einstein,100,85,88
Ramanujan,70,100,70
Gauss,100,100,70
```

```
#Print the chemistry marks scored by the students, one in each line.

level2 = open('level2.txt', 'r')
header = level2.readline()

row = level2.readline()
while row :
  fields = row.strip().split(',')
  print(int(fields[-1]))
  row = level2.readline()
level2.close()
```

```
90
88
70
70
```

We could have done this using `.readlines()` also, but this will make a list of all lines which is not good habit for a large files. because list takes too much space. so, its better to read line wise line.

OOPs

- A class name always starts with a capital letter by convention.
- The **init** method is a special method known as a constructor because it is called automatically when an object is created and is used to initialize its attributes.
- The self keyword refers to the current object, and it is used to assign values to the object’s attributes.
- Attributes defined inside the **init** method are called object attributes — they are unique to each object.
- Attributes defined outside all methods but inside the class are called class attributes — they are shared by all instances of the class.

- We can dynamically add attributes to a class or an object even after they are created.
- A dynamically created object attribute is specific to that object and cannot be accessed by other objects.
- If both an object and its class have an attribute with the same name, Python will prioritize the object attribut

```
class Student:
    counter = 0  # Class attribute

    def __init__(self, name, marks):
        self.name = name       # Object attribute
        self.marks = marks     # Object attribute
        Student.counter += 1   # Updating the class attribute

    def update_marks(self, marks):
        self.marks = marks

    def print_details(self):
        print(f'{self.name}: {self.marks}')

# Creating objects
madhavan = Student('Madhavan', 90)
print('Number of students in the program =', Student.counter)

andrew = Student('Andrew', 85)
print('Number of students in the program =', Student.counter)

usha = Student('Usha', 95)
print('Number of students in the program =', Student.counter)

# Creating attributes dynamically
madhavan.counter = -1         # Object attribute (only for 'madhavan')
Student.count = 2             # New class attribute

# Accessing attributes
print(Student.counter)        # Output: 3 (class attribute)
print(madhavan.counter)       # Output: -1 (object attribute overrides
class attribute)
print(andrew.counter)         # Output: 3 (falls back to class
attribute)
print(Student.count)          # Output: 2 (class attribute dynamically
added)
```

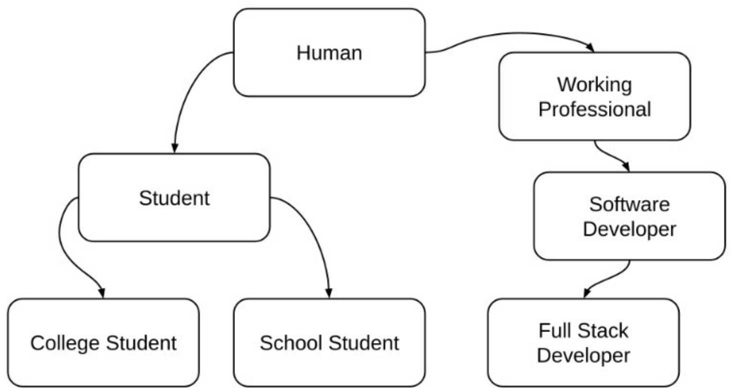
```
Number of students in the program = 1
Number of students in the program = 2
Number of students in the program = 3
3
-1
```

```
3
2
```

Inheritance

> Unity in diversity.

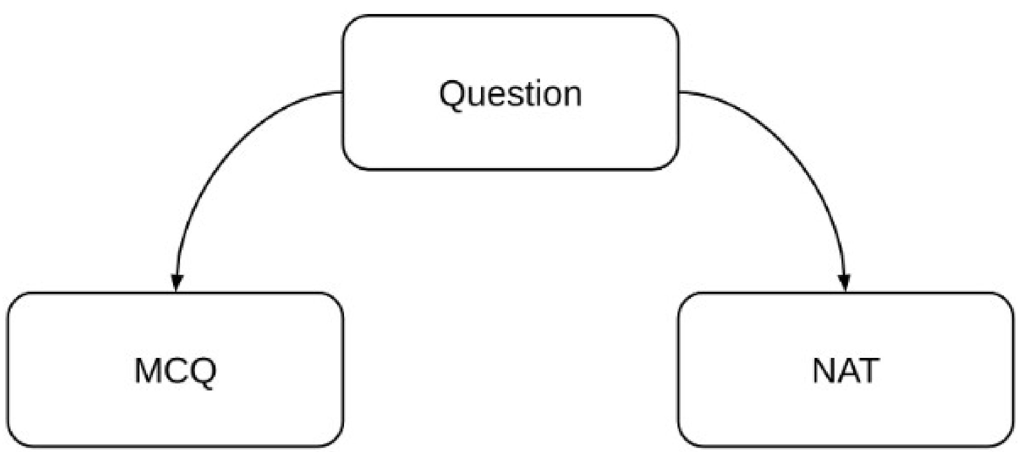
The idea of a class represents the unity, the idea of objects represent the diversity. But this diversity that we see around us is not chaotic and unordered. On the contrary, there is an organized hierarchy that we see almost everywhere around us. Consider the following image:



We humans take up different roles. Some of us are students, others are working professionals. The beauty of this program is that we have working professionals who are at the same time students. Getting back to the point, we see that there is a hierarchy. All college students are students. All students are humans. In the other branch of this directed graph, all full-stack developers are software developers, all software developers are working professionals. The basic idea behind the concept of **inheritance** is this:

> Classes that are lower in the hierarchy inherit features or attributes from their ancestors.

We would have worked on plenty of assignments across multiple courses. Each assignment is a collection of questions. Questions come in different types, some are NAT, some MCQ. So, a NAT question is not of the same type as a MCQ question. Yet, both are questions. So, we see that there is a hierarchy of relationships here:



```python
class Question:
    def __init__(self, statement, marks):
        self.statement = statement
        self.marks = marks

    def print_question(self):
        print(self.statement)

    def update_marks(self, marks):
        self.marks = marks
```

Note that we have only retained those elements as attributes that are common to all questions, irrespective of the type:

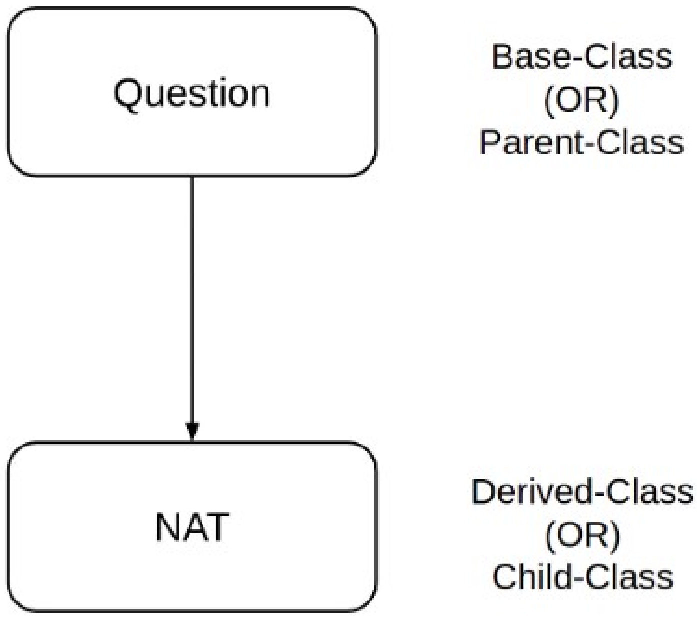
- statement of the question
- marks for the question

The next step is to define two new classes for the children of Question, one for MCQ and the other for NAT. It is here that we make use of the relationship that we just diagrammed:

```python
class NAT(Question):
    def __init__(self, statement, marks, answer):
        super().__init__(statement, marks)
        self.answer = answer

    def update_answer(self, answer):
        self.answer = answer
```

NAT is also a Question, but a specialized question. Specifically, it has an additional feature, answer, and a new method, update_answer. But all the other attributes and methods of Question are inherited by it, since NAT is just another Question.

We say that NAT is derived from Question. Question becomes the parent-class or base-class , and NAT is a child-class or derived-class.




```
#Syntex

class Derived(Base):
    def __init__(self, ...):
        pass

#### OR ####
class Child(Parent):
    def __init__(self, ...):
        ...

#Parent-child relationship
class NAT(Question):
    def __init__(self, statement, marks, answer):
        super().__init__(statement, marks)
        self.answer = answer

    def update_answer(self, answer):
        self.answer = answer


q_nat = NAT('What is 1 + 1?', 1, 2)
q_nat.update_marks(4)
print(q_nat.marks)
```

```
4
```

Method Overriding

```
class Question:
    def __init__(self, statement, marks):
        self.statement = statement
        self.marks = marks

    def print_question(self):
        print(self.statement)

    def update_marks(self, marks):
        self.marks = marks
```

Sometimes we may want to modify the behaviour of existing methods in the parent class. For example, take the case of a MCQ question. For questions of this type, the statement of a problem is incomplete without the options. The print_question method in the parent class just prints the statement, but it makes more sense to print the options as well for a MCQ question. So, we want the print_question to behave differently. Though we have inherited this method from the parent class, we can override the behaviour of the method in the following way:

```
class MCQ(Question):
    def __init__(self, statement, marks, ops, c_ops):
        super().__init__(statement, marks)
        self.ops = ops      # list of all options
        self.c_ops = c_ops  # list of correct options

    def print_question(self):
        super().print_question()
        # Assume there are only four options
        op_index = ['(a)', '(b)', '(c)', '(d)']
        for i in range(4):
            print(op_index[i], self.ops[i])

q_mcq = MCQ('What is the capital of India?',
           2,
           ['Chennai', 'Mumbai', 'Kolkota', 'New Delhi'],
           ['New Delhi'])
q_mcq.print_question()

What is the capital of India?
(a) Chennai
(b) Mumbai
(c) Kolkota
(d) New Delhi
```