



NITTE
(Deemed to be University)

**NMAM INSTITUTE
OF TECHNOLOGY**

Nitte (DU) established under Section 3 of UGC Act 1956 | Accredited with 'A+' Grade by NAAC

Introduction To Python Programming

CS1001-2



Course Outcomes:

At the end of the course student will be able to

1. Understand and experiment with the basics of python programming like data types and looping.
2. Describe the concept of functions in python for solving real-world problems.
3. Apply the Python operations for manipulating strings, lists, tuples and dictionaries.
4. Demonstrate the proficiency in handling File Systems.
5. Understand the basic knowledge on Exception handling and plotting graphs.

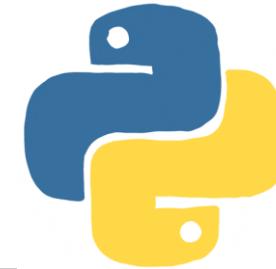
Textbooks:

1. Kenneth A. Lambert, “The Fundamentals of Python: First Programs”, Cengage Learning, 2nd Edition, 2019.
2. Yashwanth Kanetkar, Adithya Kanetkar, “Let us Python”, 3rd Edition.
3. Mark Summerfield, “Programming in Python 3 - A Complete Introduction to the Python Language”, Second Edition, Addison-Wesley, 2009.
4. Y. Daniel Liang, “Introduction to Programming Using Python”, Pearson, 2013.

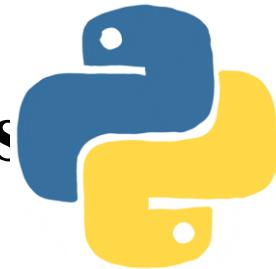
Unit -1

Introduction to Python and Function

Chapter 1:Python Concepts



Chapter 2: Design with functions



What is python?

Python is a very popular general-purpose interpreted, interactive, object-oriented, and high-level programming language.

Why to learn python?

- Python is Open Source which means its available free of cost.
- Python is simple and so easy to learn Python is versatile and can be used to create many different things.
- Python has powerful development libraries include AI, ML etc.

Python is a widely used high-level,
interpreted programming language.

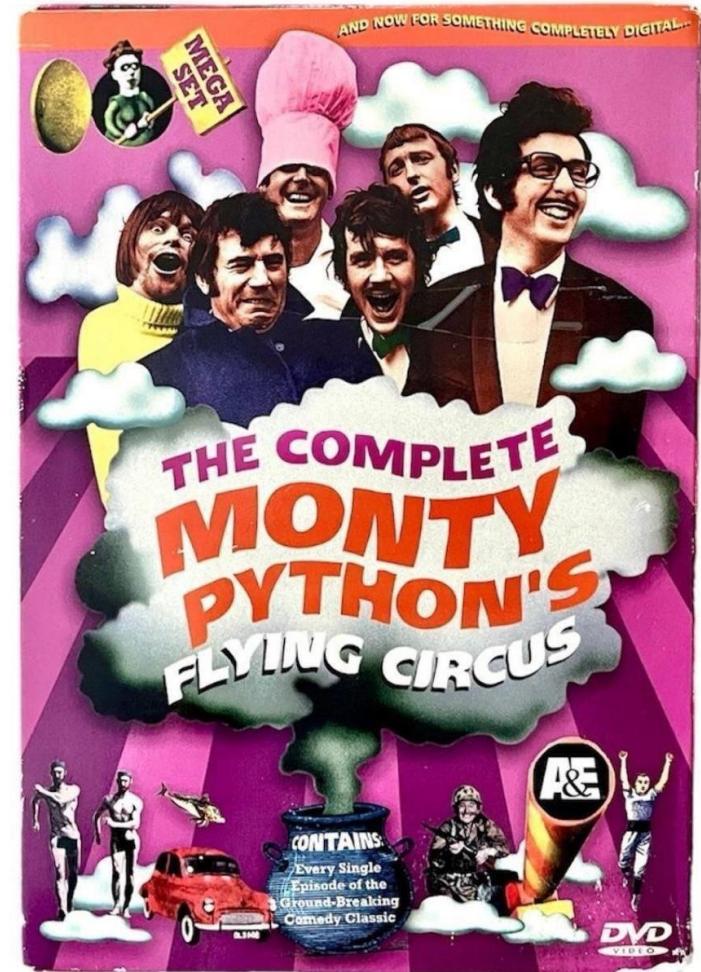
It was created by **Guido van Rossum** in
1991 and further developed by the Python
Software Foundation



Why the name python?

Guido van Rossum was interested on watching a comedy show, which is telecasting on the BBC channel from 1969 to 1974 “**The complete Monty Python’s Flying Circus**”.

Guido Van Rossum thought he needed a name that was short, unique, and slightly mysterious for his programming language, so he decided to call the language **Python**.

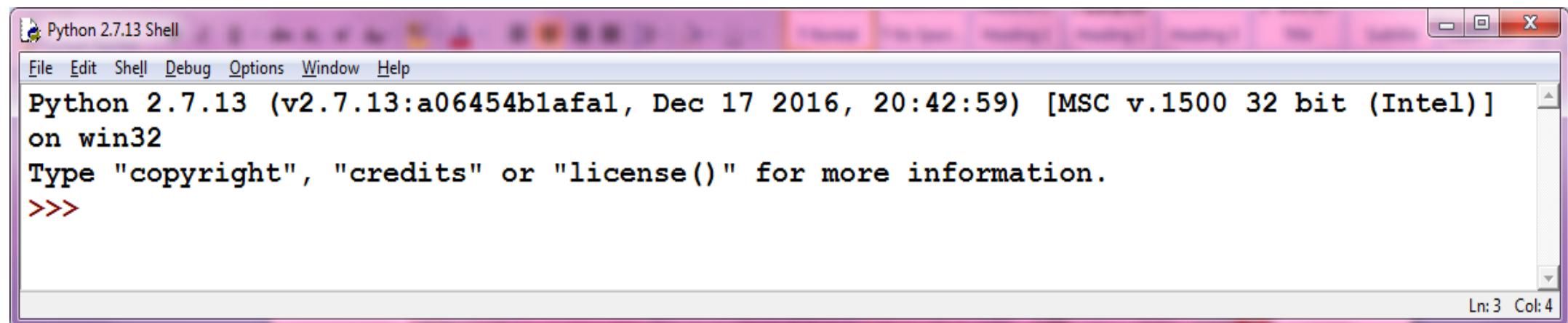


Key Features of Python

- **Python is Easy to Learn and Use:** There is no prerequisite to start Python, since it is Ideal programming language for beginners.
- **High Level Language:** Python don't let you worry about low-level details, like memory management, hardware-level operations etc.
- **Python is Interpreted:** Code is executed line-by-line directly by interpreter, and no need for separate compilation. Which means –
 - You can run the same code across different platforms.
 - You can make the changes in code without restarting the program.

a) Demonstrate about Basics of Python Programming.

- Python comes with an interactive interpreter.
- When you type **python** in your shell or command prompt, the python interpreter becomes active with a **>>>** and waits for your commands.



The screenshot shows a Windows application window titled "Python 2.7.13 Shell". The window has a menu bar with File, Edit, Shell, Debug, Options, Window, and Help. The main area displays the Python interpreter's welcome message:

```
Python 2.7.13 (v2.7.13:a06454b1afaf, Dec 17 2016, 20:42:59) [MSC v.1500 32 bit (Intel)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>>
```

In the bottom right corner of the window, there is a status bar with "Ln: 3 Col: 4".

Python 2.7.13 Shell

File Edit Shell Debug Options Window Help

```
Python 2.7.13 (v2.7.13:a06454b1afaf1, Dec 17 2016, 20:42:59) [MSC v.1500 32 bit (Intel)]
on win32
Type "copyright", "credits" or "license()" for more information.

>>> print "hello"
hello
>>> print("hello")
hello
>>>
```

Ln: 7 Col: 4

ii. Running Python Scripts in IDLE (Integrated Development and Learning Environment):

- IDLE is the standard Python development environment.
- It's name is an acronym of "**Integrated DeveLopment Environment**".
- It works well on both Unix and Windows platforms.
- It has a Python shell window, which gives you access to the Python interactive mode.

- Goto File menu click on New File (CTRL+N) and write the code and save add.py

iii. Running Python scripts in Command Prompt:

- Before going to run **python27** folder in the command prompt.
- Open your text editor, type the following text and save it as **hello.py**.

```
print "hello"
```
- In the command prompt, and run this program by calling **python hello.py**.
- Make sure you change to the directory where you saved the file before doing it.

C:\Windows\system32\cmd.exe

C:\Python27>python Hello.py
Hello, Mothilal

C:\Python27>

Python Identifier

- Python Identifier is the name we give to identify a variable, function, class, module or other object.
- That means whenever we want to give an entity a name, that's called identifier.

Python Keywords

- Python keywords are the words that are reserved.
- That means you can't use them as name of any entities like variables, classes and functions.
- **There were 33 keywords in Python 3.7. However, the number increased to 35 in Python 3.8.**

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

Keyword	Description	Keyword	Description
and	A logical operator	from	To import specific parts of a module
as	To create an alias	global	To declare a global variable
assert	For debugging	if	To make a conditional statement
break	To break out of a loop	import	To import a module
class	To define a class	in	To check if a value is present in a list, tuple, etc.
continue	To continue to the next iteration of a loop	is	To test if two variables are equal
def	To define a function	lambda	To create an anonymous function
del	To delete an object	None	Represents a null value
elif	Used in conditional statements, same as else if	nonlocal	To declare a non-local variable
else	Used in conditional statements	not	A logical operator
except	Used with exceptions, what to do when an exception occurs	or	A logical operator
False	Boolean value, result of comparison operations	pass	A null statement, a statement that will do nothing
finally	Used with exceptions, a block of code that will be executed no matter if there is an exception or not	for	To create a for loop

Variable

- A variable, as the name indicates is something whose value is changeable over time.
- In fact a variable is a memory location where a value can be stored. Later we can retrieve the value to use.
- Python has no command for declaring a variable.
- A variable is created the moment you first assign a value to it.
- Example

```
x = 5  
y = "John"  
print(x)  
print(y)
```

- No need to define type of variable.
- During the execution, the type of a variable is inferred from the context in which it is being used. Hence Python is called **dynamically-typed language**.

- Simple variable assignment

```
a = 10  
pi = 3.14  
name = 'Sanjay'
```

```
x = 4          # x is of type int  
x = "nitte"    # x is now of type str  
print(x)
```

```
x = str(3)    # x will be '3'  
y = int(3)    # y will be 3  
z = float(3)  # z will be 3.0
```

```
x = 5  
y = "John"  
print(type(x))  
print(type(y))
```

Multiple variable assignment

```
a = 10 ; pi = 31.4 ; name = 'Sanjay' # use ; as statement separator  
a, pi, name = 10, 3.14, 'Sanjay'  
a = b = c = d = 5
```

Rules for Writing Variable / Identifiers

There are some rules for writing Identifiers. But first you must know Python is case sensitive. That means **Name** and **name** are two different identifiers in Python. Here are some rules for writing Identifiers in python.

1. Identifiers can be combination of uppercase and lowercase letters, digits or an underscore (_).
So **myVariable**, **variable_1**, **variable_for_print** all are valid python identifiers.
2. An Identifier can not start with digit. So, while **variable1** is valid, **1variable** is not valid.
3. We can't use special symbols like !,#,@,%,\$ etc in our Identifier.
4. Identifier can be of any length.

Whether the following statement is true?

x, y, z = “apple”, “banana”, “Cherry”

print(x)

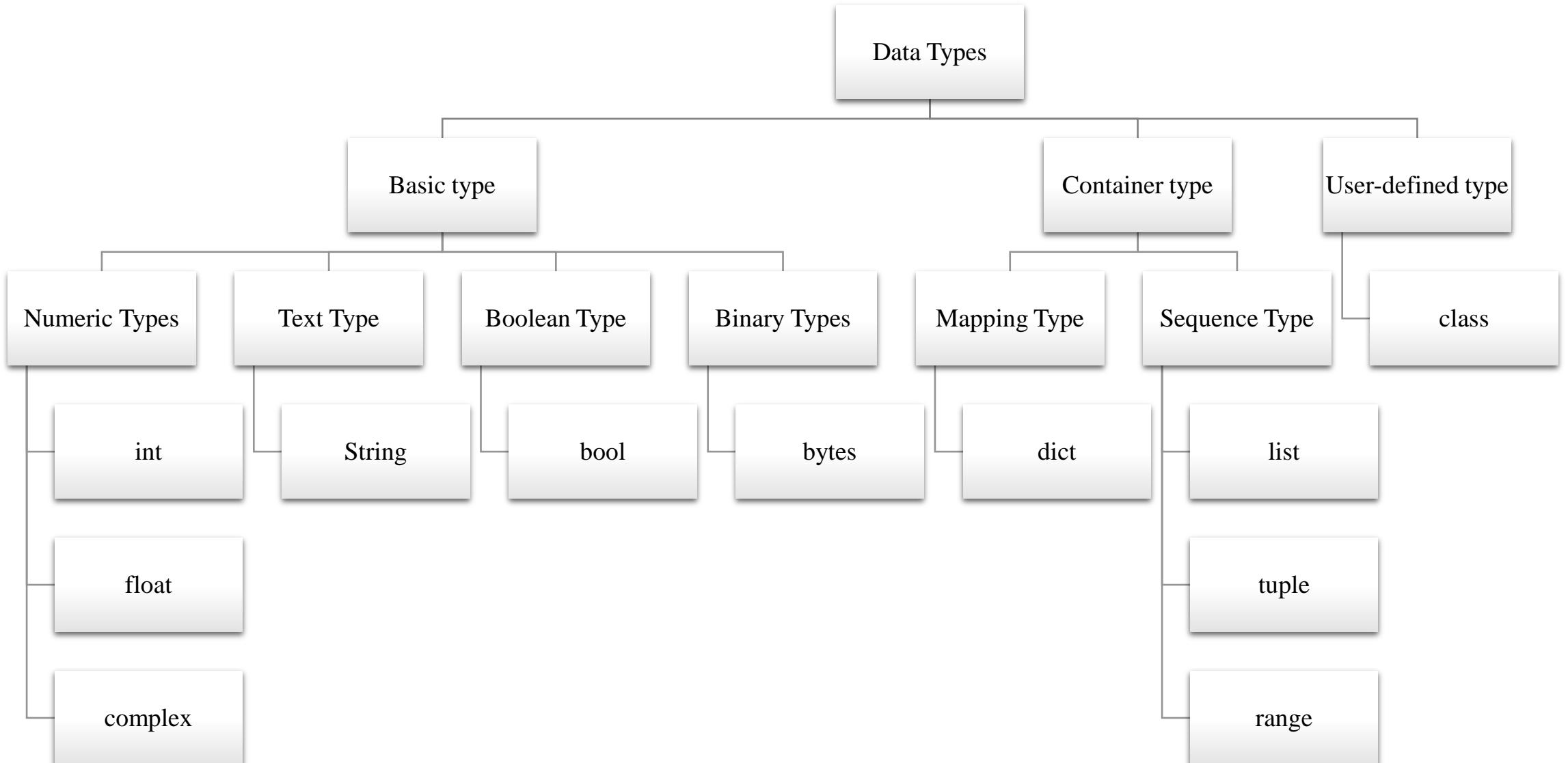
print(y)

print(z)

Python Data Types

- Variables can store data of different types, and different types can do different things.
- Python has the following data types built-in by default, in these categories:
- 3 categories of data types:
 1. Basic types: int, float, complex, bool, string, bytes
 2. Container types: list, tuple, set, dict
 3. User-defined types: Class

Categories	
Text Type:	str
Numeric Types:	int, float, complex
Sequence Types:	list, tuple, range
Mapping Type:	dict
Boolean Type:	bool
Binary Types:	bytes, bytearray
None Type:	NoneType



- `int` can be expressed in binary, decimal, octal, hexadecimal.

Integer type	Starts with	Example
decimal	0-9	156
binary	0b/0B	0b101
octal	0o/0O	0o432
hexadecimal	0x/0X	0x443

- `int` can be of any arbitrary size
- Python has arbitrary precision integers. Hence, we can create as big integers as we want.
- Arithmetic operations can be performed on integers without worrying about overflow/underflow.

`a = 123`

`b = 1234567890`

`c = 123456789012345678901234567890`

- **float** can be expressed in fractional or exponential form
 - Eg: -314.1528, 3.141528e2, 3.141528E2
- **complex** contains real and imaginary part
 - 3+2j, 1+4J
- **bool** can take any of 2 Boolean values both starting in caps
 - True, False
- **string** is an immutable collection of Unicode characters enclosed within ‘ ‘, “ ” or “”” “””
 - ‘hi’, “hi”, “””hi”””
- **bytes** represent binary data
 - B’\xa1\xe4\x56’ -represents 3bytes with hex value a1e456

Input and Output

- **print()**- used to output values on the screen
- **input()** built-in function can be used to retrieve input values from keyboard.
- **input()** function returns a string
 - i.e if 23 is entered it returns '23'
- So to perform arithmetic operations on the number entered . Convert string to int or float.

```
n = input('Enter your name: ')
age = int(input('Enter your age: '))
salary = float(input('Enter your salary: '))
print(name, age, salary)
```

Operators

Python divides the operators in the following groups:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Identity operators
- Membership operators
- Bitwise operators

1.Arithmetic operators

Operator	Name	Example
+	Addition	$x + y$
-	Subtraction	$x - y$
*	Multiplication	$x * y$
/	Division	x / y
%	Modulus	$x \% y$
**	Exponent	$x ** y$
//	Floor division	$x // y$

Operation Nuances

- On performing floor division, a/b , result is largest integer which is less than or equal to the quotient.

```
print(10 // 3)      # yields 3
print(-10 // 3)     # yields -4
print(10 // -3)     # yields -4
print(-10 // -3)    # yields 3
print(3 // 10)       # yields 0
print(3 // -10)      # yields -1
print(-3 // 10)      # yields -1
print(-3 // -10)     # yields 0
```

In $-10 // 3$, multiple of 3 which will yield -10 is -3.333 , whose floor value is -4 .

In $10 // -3$, multiple of -3 which will yield 10 is -3.333 , whose floor value is -4 .

In $-10 // -3$, multiple of -3 which will yield -10 is 3.333 , whose floor value is 3 .

2. Comparison Operators

- These are used to compare two operands. For example, compare the ages of two persons, compare the prices of two or more items, etc.
- They result in either a TRUE (Non-zero) value or FALSE (Zero) value

Operator	Name	Example
<code>==</code>	Equal	<code>x == y</code>
<code>!=</code>	Not equal	<code>x != y</code>
<code>></code>	Greater than	<code>x > y</code>
<code><</code>	Less than	<code>x < y</code>
<code>>=</code>	Greater than or equal to	<code>x >= y</code>
<code><=</code>	Less than or equal to	<code>x <= y</code>

3.Logical operators

- The result of these operators is either TRUE (ONE) or FALSE (ZERO)

Operator	Description	Example
and	Returns True if both statements are true	$x < 5$ and $x < 10$
or	Returns True if one of the statements is true	$x < 5$ or $x < 4$
not	Reverse the result, returns False if the result is true	<code>not(x < 5 and x < 10)</code>

4. Identity operators

Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location:

Operator	Description	Example
is	Returns True if both variables are the same object	x is y
is not	Returns True if both variables are not the same object	x is not y

```
x = ["apple", "banana"]  
y = ["apple", "banana"]  
z = x
```

- `print(x is z)` # returns True because z is the same object as x
- `print(x is y)` # returns False because x is not the same object as y, even if they have the same content
- `print(x == y)` # to demonstrate the difference between "is" and "==": this comparison returns True because x is equal to y

5. Membership operators

- Membership operators are used to test if a sequence is presented in an object

Operator	Description		Example
in	Returns True if a sequence with the specified value is present in the object	x in y	x = ["apple", "grapes"] print("apple" in x) Output: TRUE
not in	Returns True if a sequence with the specified value is not present in the object	x not in y	x = ["apple", "grapes"] print("pineapple" not in x) Output: TRUE

6. Bitwise operators

- Bitwise operators are used to compare (binary) numbers:

Operator	Name	Description
&	AND	Sets each bit to 1 if both bits are 1
	OR	Sets each bit to 1 if one of two bits is 1
^	XOR	Sets each bit to 1 if only one of two bits is 1
~	NOT	Inverts all the bits
<<	left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off
>>	right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off

- Take two bit values X and Y, where $X = 7 = (111)_2$ and $Y = 4 = (100)_2$.
Take Bitwise and of both X & Y
- **Note:** Here, $(111)_2$ represent binary number.

$$\begin{array}{r}
 111_2 \\
 \& 100_2 \\
 \hline
 100_2 = 4
 \end{array}$$

- Take two bit values X and Y, where $X = 7 = (111)_2$ and $Y = 4 = (100)_2$.
Take Bitwise OR of both X, Y

$$\begin{array}{r}
 111_2 \\
 | 100_2 \\
 \hline
 111_2 = 7
 \end{array}$$

Take two bit values X and Y, where $X = 7 = (111)_2$ and $Y = 4 = (100)_2$.
Take Bitwise and of both X & Y

$$\begin{array}{r} 111_2 \\ \wedge 100_2 \\ \hline 011_2 = 3 \end{array}$$

Take two bit values X and Y, where $X = 5 = (101)_2$. Take Bitwise NOT of X.

$$\begin{array}{r} \sim 1001_2 \\ \hline 0110_2 = 6 \end{array}$$

- **Example 1:**

$a = 5 = 0000\ 0101$ (Binary)

$a \ll 1 = 0000\ 1010 = 10$

$a \ll 2 = 0001\ 0100 = 20$

Example 2:

$b = -10 = 1111\ 0110$ (Binary)

$b \ll 1 = 1110\ 1100 = -20$

$b \ll 2 = 1101\ 1000 = -40$

Example 1:

$a = 10 = 0000\ 1010$ (Binary)

$a \gg 1 = 0000\ 0101 = 5$

Example 2:

$a = -10 = 1111\ 0110$ (Binary)

$a \gg 1 = 1111\ 1011 = -5$

7. Assignment operators

- Assignment operators are used to assign values to variables

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3

Operator	Example	Same As
&=	x &= 3	x = x & 3
=	x = 3	x = x 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

Precedence and Associativity

- Precedence(priority)-multiple operators in an expression
- Associativity- operators with same precedence
- Left to right associativity or right to left associativity
- $c=a*b/c$
 - Same precedence
 - * is done before / (arithmetic operators have Left to right associativity)

Description	Operator	Associativity
Grouping	()	Left to Right
Function call	function()	Left to Right
Slicing	[start:end:step]	Left to Right
Exponentiation	**	Right to Left
Bitwise NOT	~	Right to Left
Unary plus / minus	+ -	Left to Right
Multiplication	*	Left to Right
Division	/	Left to Right
Modular Division	%	Left to Right
Addition	+	Left to Right
Subtraction	-	Left to Right
Bitwise left shift	<<	Left to Right
Bitwise right shift	>>	Left to Right
Bitwise AND	&	Left to Right
Bitwise XOR	^	Left to Right
Bitwise OR		Left to Right
Membership	in not in	Left to Right
Identity	is is not	Left to Right
Relational	< > <= >=	Left to Right
Equality	==	Left to Right
Inequality	!= <>	Left to Right
Logical NOT	not	Left to Right
Logical AND	and	Left to Right
Logical OR	or	Left to Right
Assignment	= += -= *= /= %= //*= **= &= = ^= >>= <<=	Right to Left

Type Conversions

- Mixed mode operations:
 - Operation between int and float will yield float.
 - Operation between int and complex will yield complex.
 - Operation between float and complex will yield complex.
- We can convert one numeric type to another using built-in functions.
 - **int()** , **float()**, **complex()**, **bool()**

- `int()` - constructs an integer number from an integer literal, a float literal (by removing all decimals), or a string literal (providing the string represents a whole number)

```
# x will be 1  
x = int(1)  
# y will be 2  
y = int(2.8)  
# z will be 3  
z = int("3")
```

- `float()` - constructs a float number from an integer literal, a float literal or a string literal (providing ~~the string represents a float or an integer~~)

```
# y will be 2.8  
x = float(1)  
# z will be 3.0  
y = float(2.8)  
# w will be 4.2  
z = float("3")  
w = float("4.2")
```

- `str()` - constructs a string from a wide variety of data types, including strings, integer literals and float literals

- `x = str("s1") # x will be 's1'`

`y = str(2) # y will be '2'`

`z = str(3.0) # z will be '3.0'`

Conversion	Explanation
<code>int(float/numeric string)</code>	from float/numeric string to int
<code>int(numeric string, base)</code>	from numeric string to int in base
<code>float(int/numeric string)</code>	from int/numeric string to float
<code>float (int)</code>	from int to float
<code>complex(int/float)</code>	Convert to complex with imaginary part zero
<code>complex(int/float, int/float)</code>	Convert to complex
<code>bool(int/float)</code>	from int to float to True/False
<code>str(int/float/bool)</code>	Convert to string
<code>chr(int)</code>	Yields a character corresponding to int

Built-in functions

- `print()` is also a built-in function.
- Help about any built-in function is available using **help(function)**
- **Built-in function commonly used with numbers are:**

<code>abs(x)</code>	# returns absolute value of x	<code>max(x1, x2,...)</code>	# returns largest argument
<code>pow(x, y)</code>	# returns value of x raised to y	<code>divmod(x, y)</code>	# returns a pair($x // y, x \% y$)
<code>min(x1, x2,...)</code>	# returns smallest argument	<code>round(x [,n])</code>	# returns x rounded to n digits after .

<code>bin(x)</code>	# returns binary equivalent of x
<code>oct(x)</code>	# returns octal equivalent of x
<code>hex(x)</code>	# returns hexadecimal equivalent of x

```
a = abs(-3)          # assigns 3 to a  
print(min(10, 20, 30, 40))    # prints 10  
print(hex(26))        # prints 1a
```

Built-in Modules

- Each module contains many functions.
 - math, cmath, random, decimal

math - many useful mathematics functions.

cmath - functions for performing operations on complex numbers.

random - functions related to random number generation.

decimal - functions for performing precise arithmetic operations.

- Mathematical functions in **math** module:

pi, e	# values of constants pi and e
sqrt(x)	# square root of x
factorial(x)	# factorial of x
fabs(x)	# absolute value of float x
log(x)	# natural log of x (log to the base e)
log10(x)	# base-10 logarithm of x
exp(x)	# e raised to x
trunc(x)	# truncate to integer
ceil(x)	# smallest integer $\geq x$
floor(x)	# largest integer $\leq x$
modf(x)	# fractional and integer parts of x

- **round()** built-in function can round to a specific number of decimal places, whereas
- **trunc(), ceil(), floor()** always round to zero decimal places.

- Trigonometric functions in **math** module:

degrees(x)	# radians to degrees
radians(x)	# degrees to radians
sin(x)	# sine of x radians
cos(x)	# cosine of x radians
tan(x)	# tan of x radians
sinh(x)	# hyperbolic sine of x
cosh(x)	# hyperbolic cosine of x
tanh(x)	# hyperbolic tan of x
acos(x)	# cos inverse of x, in radians
asin(x)	# sine inverse of x, in radians
atan(x)	# tan inverse of x, in radians
hypot(x, y)	# sqrt(x * x + y * y)

- Random number generation functions from **random** module:

```
random( )          # random number between 0 and 1  
randint(start, stop) # random number in the range  
seed( ) # sets current time as seed for random number generation  
seed(x) # sets x as seed for random number generation logic
```

- To use functions present in a module, we need to import the module using **import** statement:

```
import math  
import random  
print(math.factorial(5))      # prints 120  
print(math.degrees(math.pi))   # prints 180.0  
print(random.random( ))        # prints 0.8960522546341796
```

- To get list of functions in each built-in module:

```
import math  
print(dir(__builtins__))    # 2 underscores before and after builtins
```

```
print(dir(math))
```

Comments in python

What are Comments?

- Comments are used to explain the code and make it easier to read.
- Comments are ignored during execution.
- **Single line comment begin with #**

```
#Program to add 2 numbers
```

```
a=b+c
```

- Multiline comment written in a pair of “” or “””

“”” This function calculates the greatest common divisor (GCD) of two numbers using the Euclidean algorithm. The GCD of two numbers is the largest number that divides both of them without leaving a remainder. “””

```
def gcd(a, b):  
    while b:  
        a, b = b, a % b
```

```
    return a
```

```
result = gcd(48, 18)
```

Indentation

- ‘Unexpected indent’

```
a = 20  
    b = 45
```

Control statements

- So far statements in all our programs got executed sequentially or one after the other.
- Sequence of execution of instructions in a program can be altered using:
 - Decision control instruction
 - Repetition control instruction

- Colon(:) after if, else, elif is compulsory.
- Statements in if block, else block, elif block have to be indented.
- Indented statements are treated as block of statements.
- Used to group statements. Use either 4 spaces or a tab for indentation.

If statement:

The if statement is used to test a specific condition. If the condition is true, a block of code (if-block) will be executed.

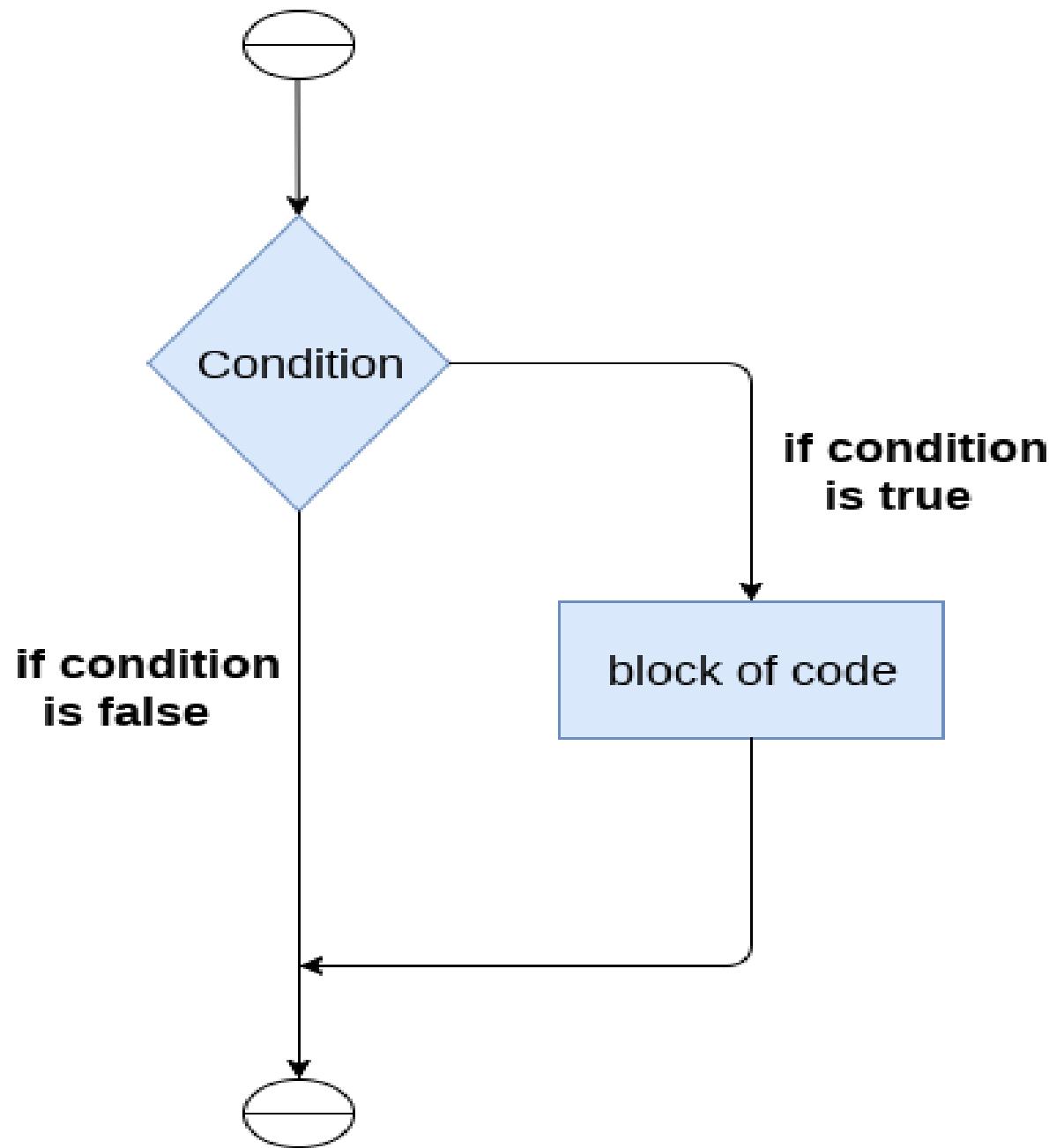
Syntax:

if condition:

 statement1

 statement2

 statement3



Example:

```
a = 30
```

```
b = 100
```

```
if b > a:
```

```
    print("b is greater than a")
```

```
print("program ends here..")
```

The if-else statement

The if-else statement is similar to if statement except the fact that, it also provides the block of the code for the false case of the condition to be checked. If the condition provided in the if statement is false, then the else statement will be executed.

Syntax:

if condition:

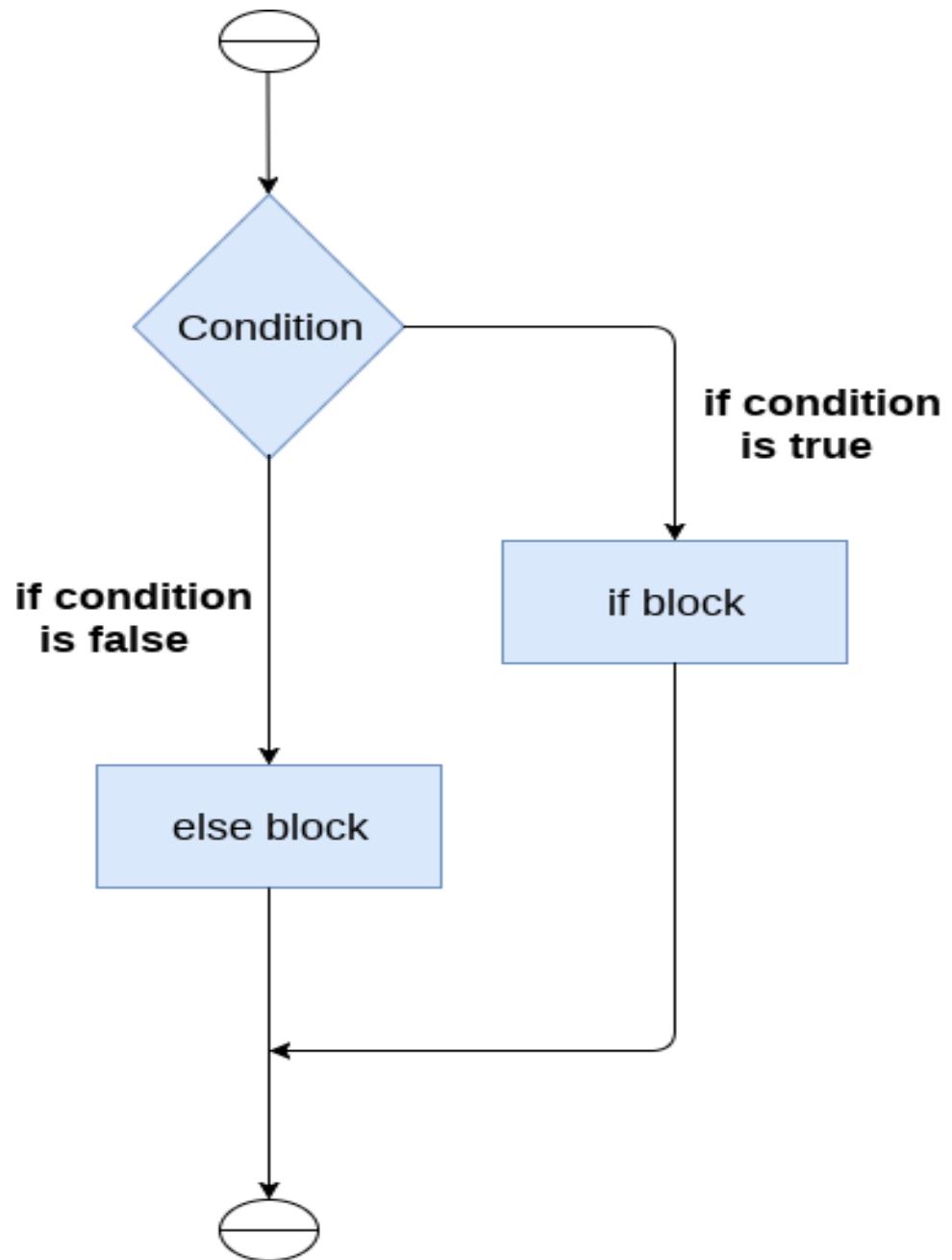
statement1

statement2

else:

statement3

statement4



Example:

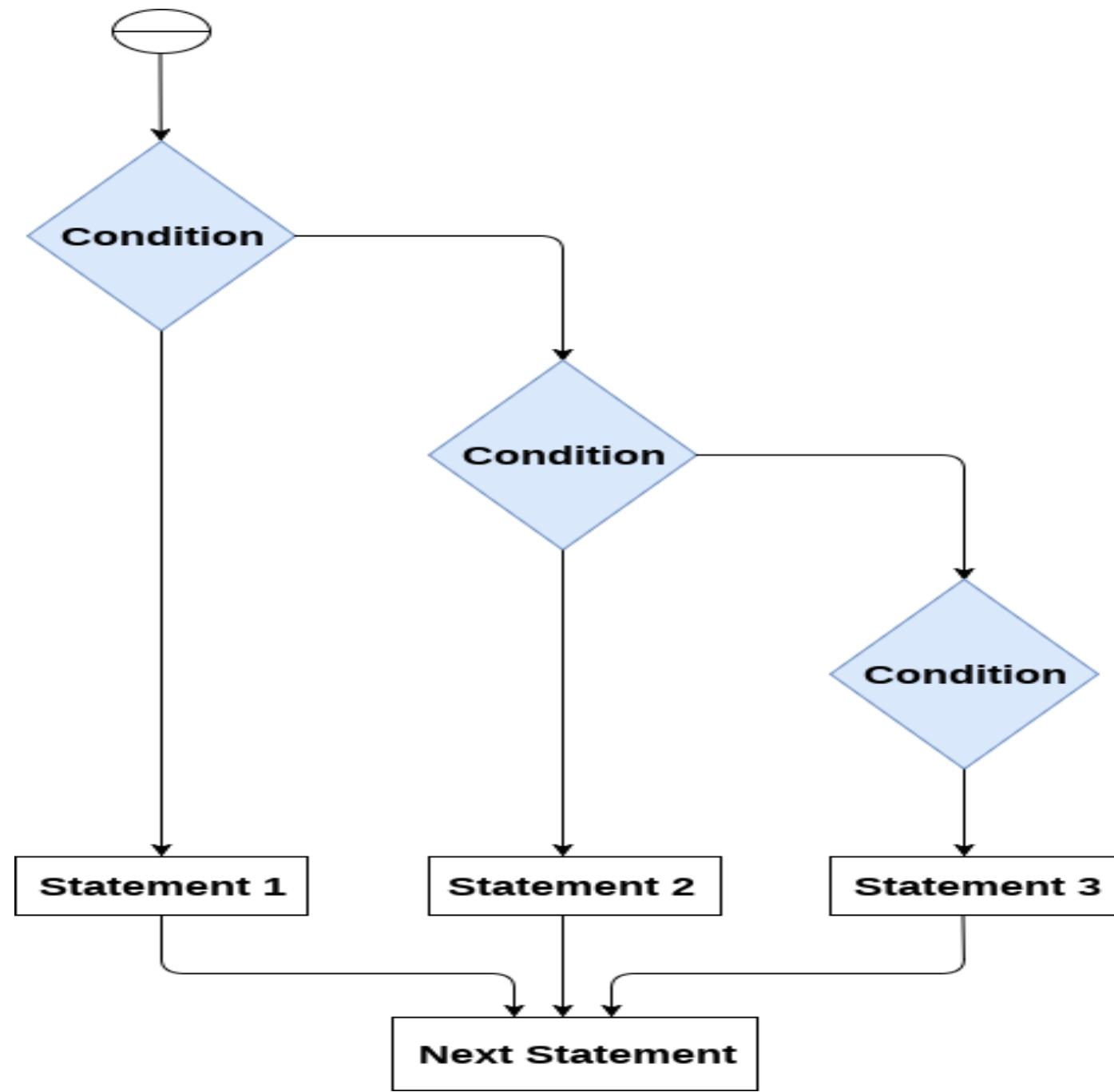
```
a = 30
b = 100
if b > a:
    print("b is greater than a")
else:
    print("b is greater than a")
print("program ends here..")
```

The elif statement:

- The elif statement enables us to check multiple conditions and execute the specific block of statements depending upon the true condition among them.
- We can have any number of elif statements in our program depending upon our need.

Syntax:

```
if condition1:  
    statement1  
    statement2  
elif condition2 :  
    statement3  
elif condition3 :  
    statement4  
else :  
    statement4  
next statement
```



Example:

a = 30

b = 100

if a > b:

 print("a is greater than b")

elif a < b:

 print("a is greater than b")

else:

 print("a and b are equal")

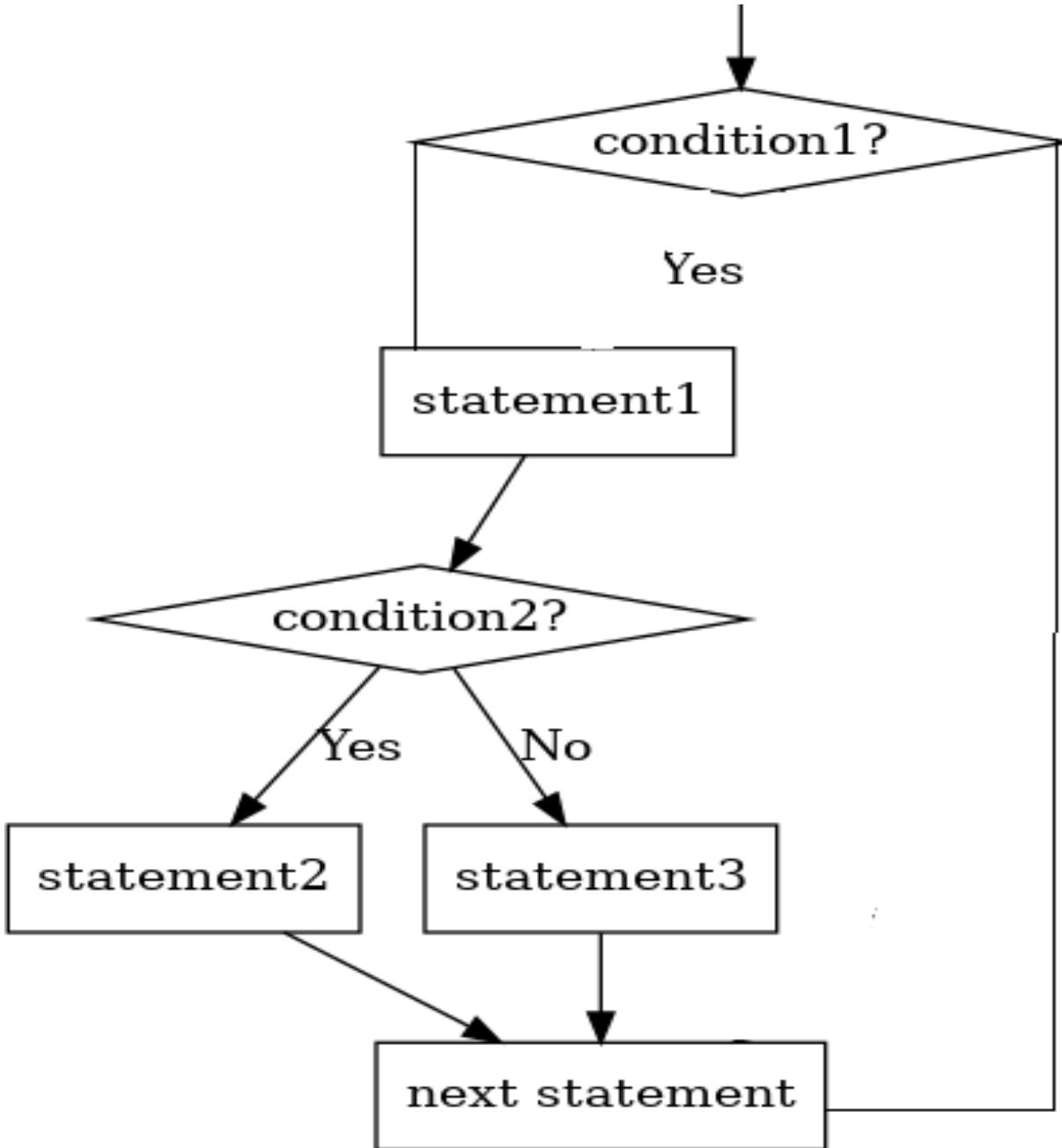
print("thank you")

Nested if:

Python supports **nested if statements** which means we can use a conditional if and if...else statement inside an existing **if statement**.

Syntax:

```
if condition1:  
    statement1  
    if condition2:  
        statement2  
    else:  
        statement3  
next statement
```



Example:

```
num = 36
```

```
print ("num = ", num)
```

```
if num % 2 == 0:
```

```
    if num % 3 == 0:
```

```
        print ("Divisible by 3 and 2")
```

```
    else:
```

```
        print ("Divisible by 2 but not by 3")
```

```
print("program ends here..")
```

Repetition Control Instruction

- 2 types of repetition control instructions:
 - while
 - for

For loop:

for is used to iterate over elements of a sequence such as string, tuple or list.

Syntax:

for value in sequence:

 loop body

2 types:

```
for var in list :  
    statement1  
    statement2
```

```
for var in list :  
    statement1  
    statement2  
else :  
    statement3  
    statement4
```

- During each iteration **var** is assigned the next value from the list.
- In place of a list, a string, a tuple, set or dictionary can also be used.
- **else** block is optional.
- **If present**, it is executed if loop is not terminated abruptly using **break**.

```
x = 'nmamit'
```

```
for i in x:
```

```
    print(i)
```

output: ??

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    print(x)
```

Output:

apple
banana
cherry

Usage of for loop

- A for loop can be used in the following two situations:
 - Repeat a set of statements a finite number of times.
 - Iterate through a string, list, tuple, set or dictionary.
- To repeat a set of statements a finite number of times a built-in function **range()** is used.

- **range()** function generates a sequence of integers:
 - range(10)- generates a number from 0 to 9
 - range(10,20)- generates a number from 10 to 19
 - range(10,20,2)- generates a number from 10 to 19 in step of 2
 - range(20,10,-3)- generates a number from 20 to 9 in step of -3
- **range()** cannot generate sequence of floats.
 - **range(start, stop, step)**
- Produces a sequence of integers from start(inclusive) to stop(exclusive) by step.

- **Example 1:**

```
for x in range(2, 30, 3):  
    print(x)
```

Output: 2 5 8 11 14 17 20 23 26 29

- Example 2:**

```
for i in range(5, -1, -1):  
    print(i)
```

Output: 5 4 3 2 1 0

- The list of numbers generated using `range()` can be iterated through using a for loop.

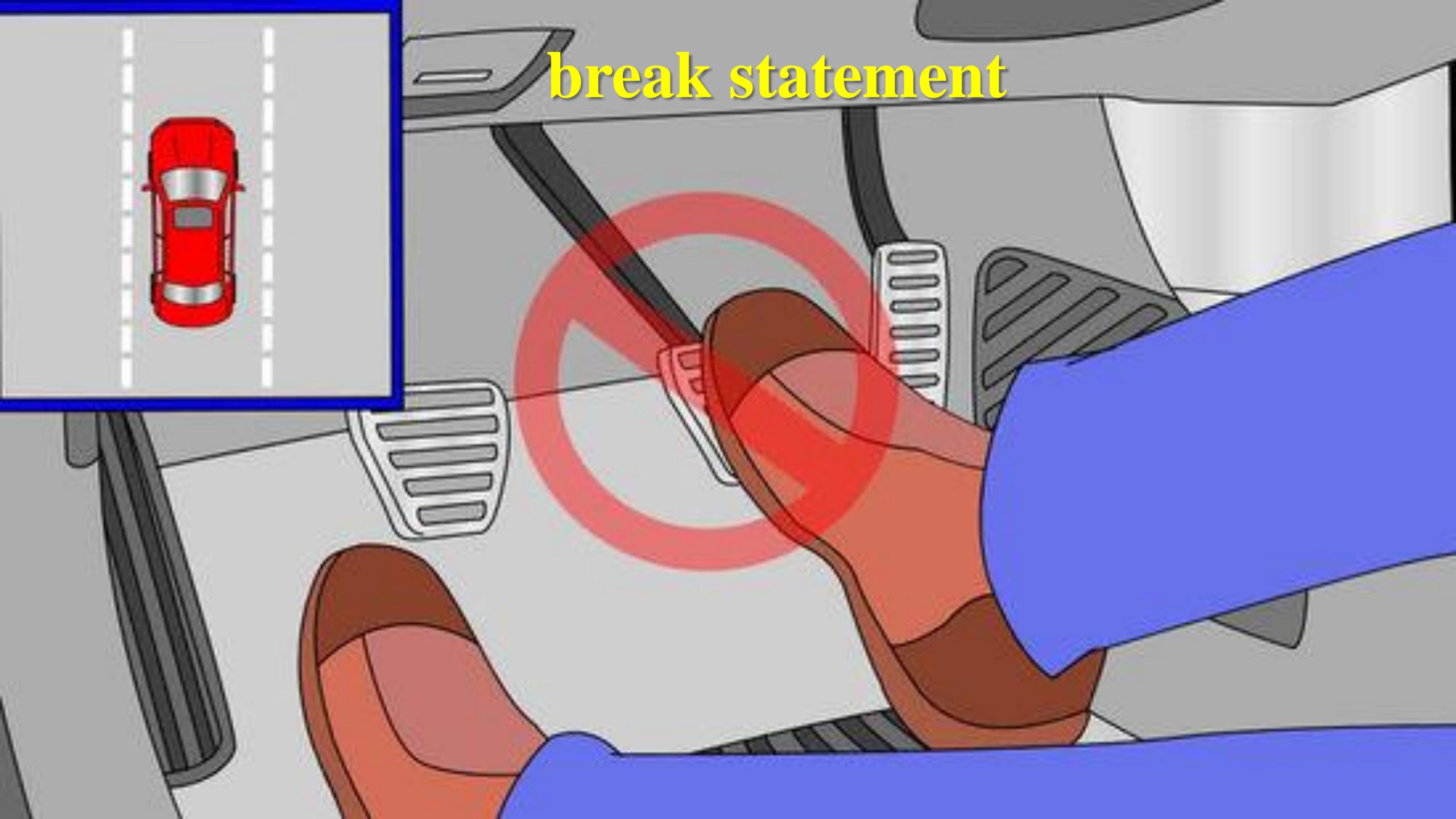
```
for i in range(1, 10, 2):  
    print(i, i * i, i * i * i)
```

- Output:

1	1	1
3	9	27
5	25	125
7	49	343
9	81	729

- `for` can be used to iterate through a string, list, tuple, set or dictionary as shown below:

```
for char in 'Leopard' :  
    print(char)  
for animal in ['Cat', 'Dog', 'Tiger', 'Lion', 'Leopard'] :  
    print(animal)  
for flower in ('Rose', 'Lily', 'Jasmine') :  
    print(flower)  
for num in {10, 20, 30, -10, -25} :  
    print(num)  
for key in {'A101' : 'Rajesh', 'A111' : 'Sunil', 'A112' : 'Rakesh'} :  
    print(key)
```



break statement

The break Statement

The break statement in Python is used to exit or “break” out of a loop (either a for or while loop) prematurely before the loop has iterated through all its items or reached its condition.

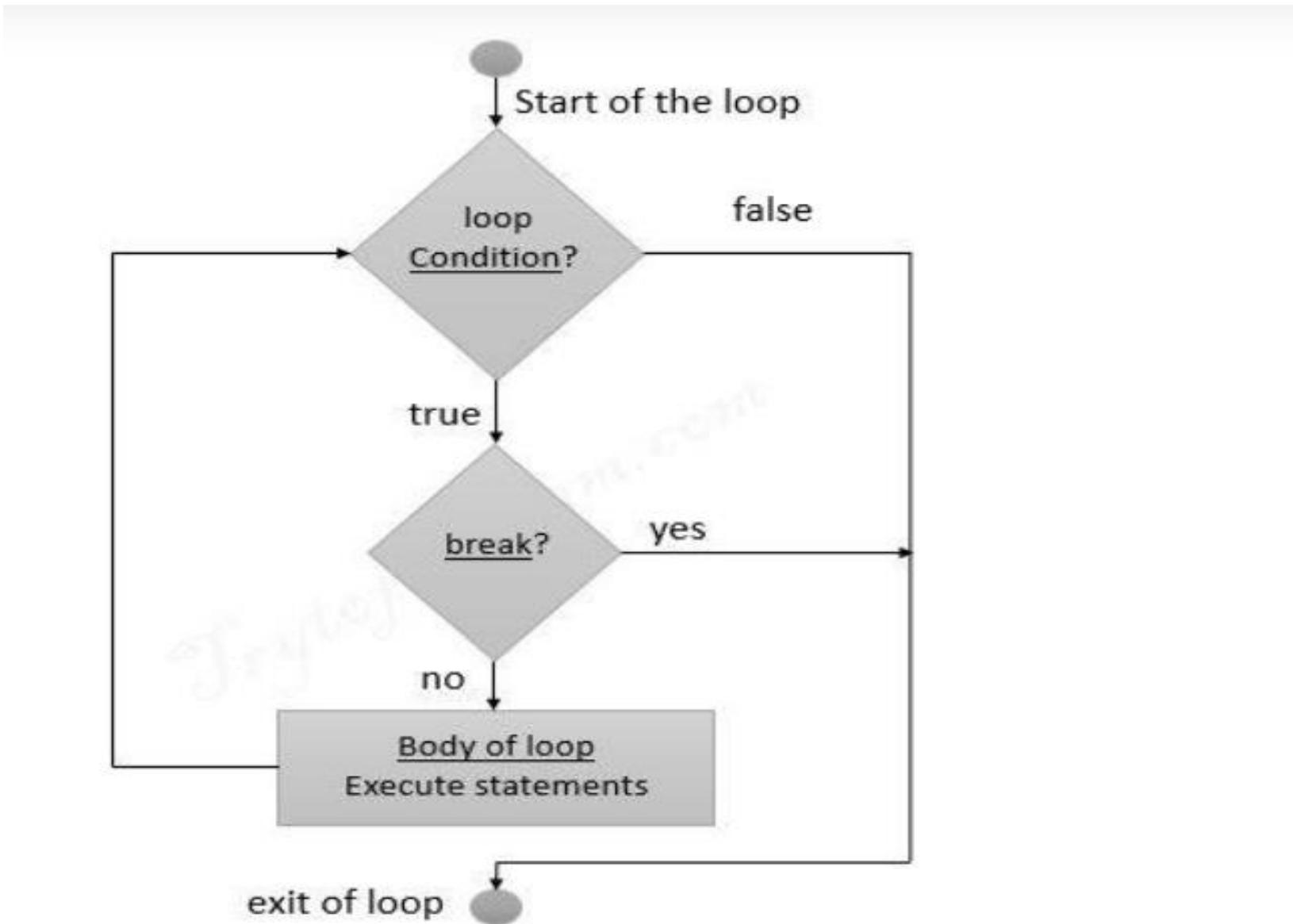
Example:

```
x = 'nmamit'  
for i in x:  
    if i=='i':  
        break  
    print(i)
```

```
for i in range(5):  
    if i == 3:  
        break  
    print(i)
```

Output:

Working of break statement



The continue Statement

With the continue statement we can stop the current iteration of the loop, and continue with the next.

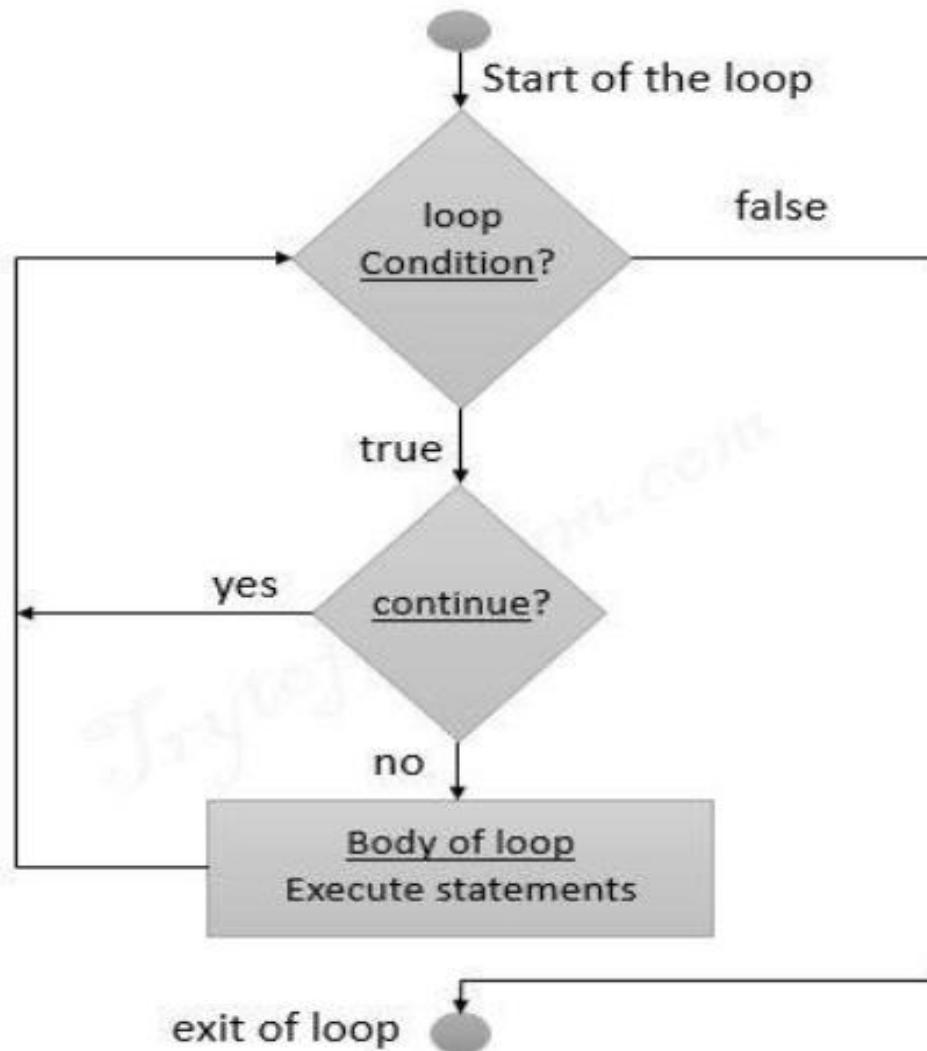
Python **continue statement** is used to skip the execution of the program block and returns the control to the beginning of the current loop to start the next iteration.

```
x = 'nmamit'  
for i in x:  
    if i=='i':  
        continue  
    print(i)
```

```
for i in range(5):  
    if i == 3:  
        continue  
    print(i)
```

Output:

Working of continue statement:



Using else Statement with for Loop

- A loop expression and an else expression can be connected in Python.
- After the circuit has finished iterating over the list, the else clause is combined with a for Loop.

Example:

```
for x in range(6):
    if x == 3:
        break
        print(x)
else:
    print("finished!")
```

```
start = int(input("Enter starting range"))  
end = int(input("Enter ending range"))  
c=0  
print("Prime numbers between", start, "and", end, "are:")  
  
for num in range(start, end+1):  
    # all prime numbers are greater than 1  
    if num > 1:  
        for i in range(2, num):  
            if (num % i) == 0:  
                break  
        else:  
            c=c+1  
            print(num)  
if(c==0):  
    print("There are no prime numbers in the range")
```

while loop

A while loop in Python is a control flow statement used to repeat a block of code as long as a specified condition is True. It's especially useful when the number of iterations isn't known beforehand but depends on some dynamic condition.

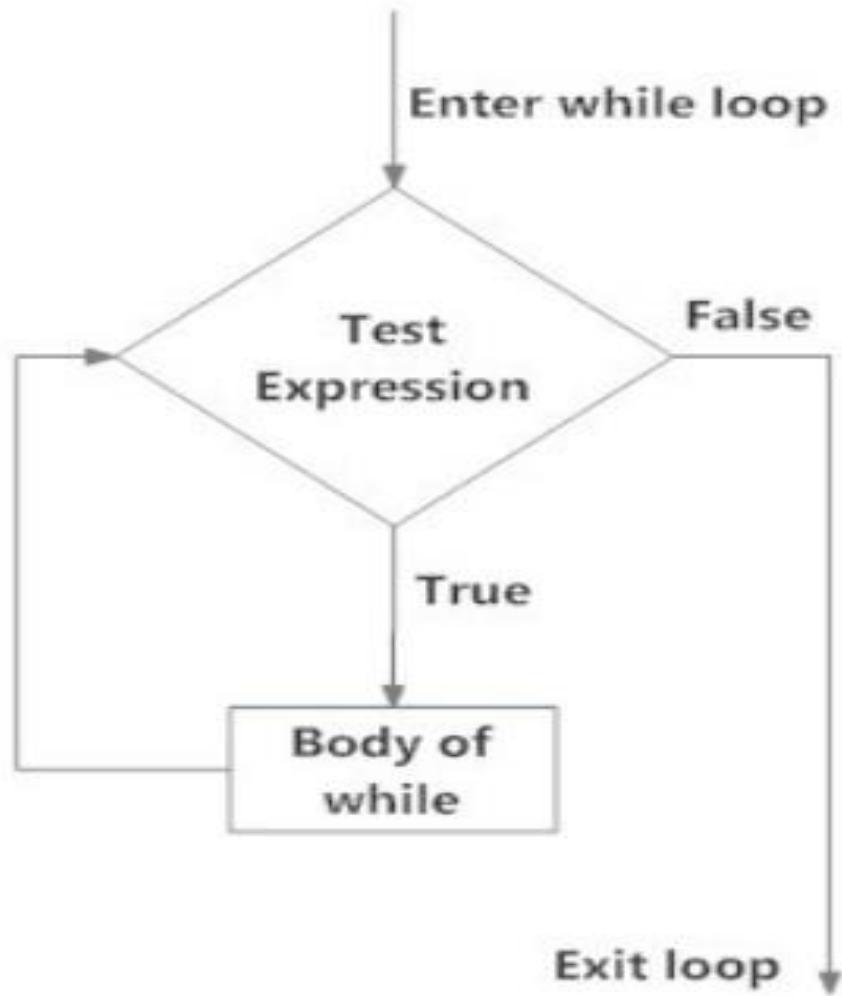
Syntax:

while expression:

 statement(s)

- **condition:** This is a boolean expression. If it evaluates to True, the code inside the loop will execute.
- **statement(s):** These are the statements that will be executed during each iteration of the loop.

Working of while loop



Snippet 1: Print Numbers from 1 to 5

```
num = 1
while num <= 5:
    print(num)
    num += 1
```

Output:

1
2
3
4
5

Snippet 2: Sum of Numbers from 1 to 10

```
num=1
total_sum=0

while num<=10:
    total_sum += num
    num += 1
print("The sum of numbers from 1 to 10 is:", total_sum)
```

Snippet 3: Even Numbers Between 1 and 10

```
num = 1
print("Even numbers between 1 and 10 are:")
while num <= 10:
    if num % 2 == 0:
        print(num)
    num += 1
```

Snippet 4: Print "Hello, World!" 5 Times using while

```
count=0  
while count < 5:  
    print("hello world")  
    count += 1
```

```
hello world  
hello world  
hello world  
hello world  
hello world
```

Snippet 6: Infinite Loop

```
while True:  
    print("This will run forever!")
```

Output:

The string “This will run forever!” will be printed infinite times.

Syntax:

- Usage of break with for loop
- Usage of break with while loop

```
for val in sequence:  
    # code  
    if condition:  
        break
```



```
# code
```

```
while condition:  
    # code  
    if condition:
```



```
        break
```

```
# code
```

Syntax:

- Usage of continue with for loop

```
for val in sequence:  
    # code  
    if condition:  
        continue  
  
    # code
```

- Usage of continue with while loop

```
while condition:  
    # code  
    if condition:  
        continue  
  
    # code
```

while loop with else

When the condition becomes false, the statement immediately after the loop is executed. The else clause is only executed when your while condition becomes false. If you break out of the loop, or if an exception is raised, it won't be executed

```
i = 0
while i < 4:
    i += 1
    print(i)
else: # Executed because
      no break in for
    print("No Break\n")
```

```
i = 0
while i < 4:
    i += 1
    print(i)
    break
else: # Not executed as
      there is a break
    print("No Break")
```

pass statement

The Python pass statement is a null statement. But the difference between pass and comment is that comment is ignored by the interpreter whereas pass is not ignored.

pass

Syntax:

pass

What is pass statement in Python?

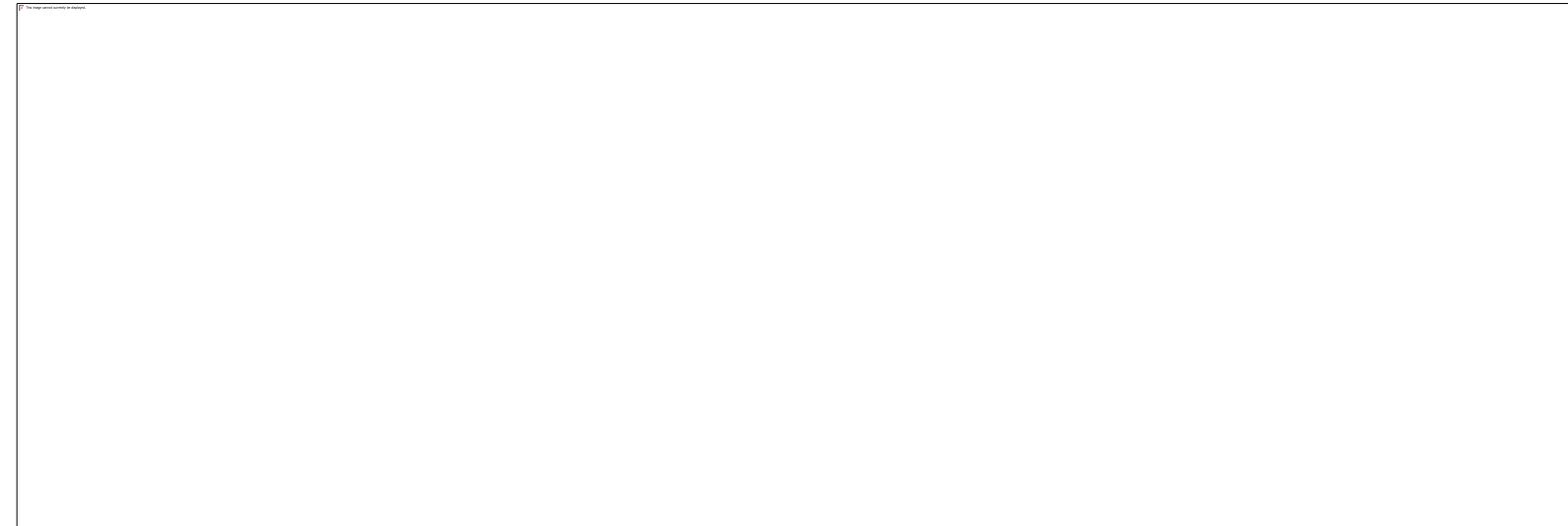
When the user does not know what code to write, So user simply places a pass at that line. Sometimes, the pass is used when the user doesn't want any code to execute. So users can simply place a pass where empty code is not allowed, like in loops, function definitions, class definitions, or in if statements. So using a pass statement user avoids this error.

Using pass With Conditional Statement

Output

```
n = 10
n=n+1
# use pass inside if statement
if n > 9:
    pass
else:
    print("Hello")
```

Using pass With Conditional Statement



Output

Hello

==> Code Execution Successful ==>

```
n = int(input('Enter a number: '))
if n > 0:
    flag = True
    print(n * n)
elif n < 0:
    flag = True
    print(n * n * n)
```

```
n = int(input('Enter a number: '))
if n > 0:
    flag = True
    print(n * n)
elif n < 0:
    flag = True
    print(n * n * n)
else:
    pass # does nothing on execution
```

Answer

- This is misleading code. At a later date, anybody looking at this code may feel that `flag = True` should be written outside `if` and `else`.
- Better code will be as follows:

Using pass With Looping Statement

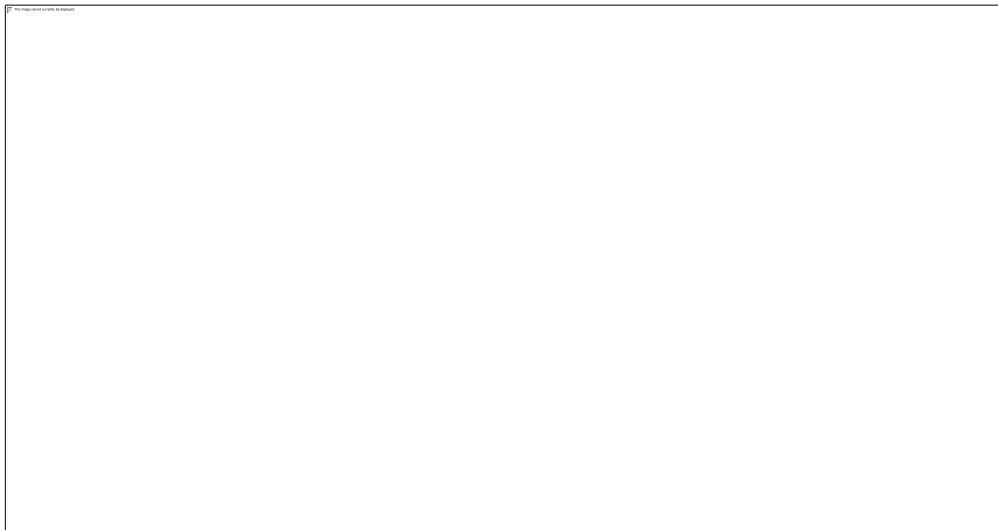
```
n = 10
for i in range(n):
    # pass can be used as placeholder
    # when code is to added later
    pass
print(i)
```

Output

9

== Code Execution Successful ==

Using pass With Looping Statement (while)



FUNCTIONS:

Function is a sub program which consists of set of instructions used to perform a specific task. A large program is divided into basic building-blocks called function.

- **Need For Function:**

- ❖ When the program is too complex and large they are divided into parts. Each part is separately coded and combined into single program. Each subprogram is called as function.
- ❖ Debugging, Testing and maintenance becomes easy when the program is divided into subprograms.
- ❖ Functions are used to avoid rewriting same code again and again in a program.
- ❖ Function provides code re-usability
- ❖ The length of the program is reduced.

- **Types of function:**

Functions can be classified into two categories:

- i) Built in function
- ii) user defined function

i) Built in functions

- ❖ Built in functions are the functions that are already created and stored in python.

These built in functions are always available for usage and accessed by a programmer. It cannot be modified.

Built in function	Description
>>>max(3,4) 4	# returns largest element
>>>min(3,4) 3	# returns smallest element
>>>len("hello") 5	#returns length of an object
>>>range(2,8,1) [2, 3, 4, 5, 6, 7]	#returns range of given values
>>>round(7.8) 8.0	#returns rounded integer of the given number
>>>chr(5) \x05'	#returns a character (a string) from an integer
>>>float(5) 5.0	#returns float number from string or integer
>>>int(5.0) 5	# returns integer from string or float
>>>pow(3,5) 243	#returns power of given number
>>>type(5.6) <type 'float'>	#returns data type of object to which it belongs
>>>t=tuple([4,6.0,7]) (4, 6.0, 7)	# to create tuple of items from list
>>>print("good morning") Good morning	# displays the given object
>>>input("enter name: ") enter name : George	# reads and returns the given string

ii) User Defined Functions:

- ❖ User defined functions are the functions that programmers create for their requirement and use.
 - ❖ These functions can then be **combined to form module** which can be used in other programs by importing them.
-
- ❖ Advantages of user defined functions:
 - Programmers working on large project can divide the workload by making different functions.
 - If repeated code occurs in a program, function can be used to include those codes and execute when needed by calling that function.

Function definition: (Sub program)

- ❖ def keyword is used to define a function.
- ❖ Give the function name after def keyword followed by parentheses in which arguments are given.
- ❖ End with colon (:)
- ❖ Inside the function add the program statements to be executed
- ❖ End with or without return statement

Syntax:

```
def fun_name(Parameter1,Parameter2...Parameter n):  
    statement1  
    statement2...  
    statement n  
    return[expression]
```

- **Example:**

```
def my_add(a,b):  
    c=a+b  
    return c
```

Function Calling:

- Once we have defined a function, we can call it from another function, program or even the Python prompt.
- To **call a function** we **simply type the function name with appropriate arguments.**

- Example:

x=5

y=4

my_add(x,y)

Flow of Execution:

- ❖ The order in which statements are executed is called the **flow of execution**
- ❖ Execution always begins at the **first statement of the program**
- ❖ Statements are executed one at a time, in order, from top to bottom.
- ❖ Function definitions do not alter the flow of execution of the program, but remember that statements inside the function are not executed until the function is called.
- ❖ Function calls are like a bypass in the flow of execution. Instead of going to the next statement, the flow jumps to the first line of the called function, executes all the statements there, and then comes back to pick up where it left off.

Note: When you read a program, don't read from top to bottom. Instead, follow the flow of execution. This means that you will read the **def** statements as you are scanning from top to bottom, but you should skip the statements of the function definition until you reach a point where that function is called.

Parameters And Arguments:

Parameters(Formal Parameters):

- Parameters are the value(s) provided in the parenthesis when we write function header.
- These are the values required by function to work.
- If there is more than one value required, all of them will be listed in parameter list separated by **comma**.
- Example: **def my_add(a,b):**

Arguments(Actual Parameters) :

- Arguments are the value(s) provided in function call/invoke statement.
- List of arguments should be supplied in same way as parameters are listed.
- Bounding of parameters to arguments is done 1:1, and so there should be same number and type of arguments as mentioned in parameter list.
- Example: **my_add(x,y)**

Difference between Parameter and argument:

Aspect	Parameter	Argument
Definition	A variable listed in the function definition.	A value passed to the function when it is called.
Location	Appears in the function definition.	Appears in the function call.
Purpose	Acts as a placeholder for values that will be received.	Provides actual data for function execution.
Example	def greet(name): where name is a parameter.	greet("Alice") "Alice" is an argument.
Types	Positional, Default, Variable-Length	Positional, Keyword, Default, and Variable-Length arguments.

RETURN STATEMENT:

- The **return statement** is used to exit a function and go back to the place from where it was called.
- If the return statement has no arguments, then it will not return any values. But exits from function.
- **Syntax:**
`return[expression]`

- **Example:**

```
def my_add(a,b):
```

```
    c=a+b
```

```
    return c
```

```
x=5
```

```
y=4
```

```
print(my_add(x,y))
```

- **Output:**

9

The pass Statement

The `pass` statement serves as a placeholder for future code, preventing errors from empty code blocks.

```
def function():
    pass          # this will execute without any action
or error
function()
```

ARGUMENTS TYPES:

1. Required Arguments
2. Keyword(named) Arguments
3. Default Arguments
4. Variable length Arguments

1. Required Arguments:

The number of arguments in the function call should match exactly with the function definition.

```
def my_details( name, age ):
```

```
    print("Name: ", name)
```

```
    print("Age ", age)
```

```
return
```

```
my_details("george",56)
```

- Output:**

Name: george

Age 56

2. Keyword(named) Arguments:

- Python interpreter is able to use the keywords provided to match the values with parameters even though if they are arranged in out of order.

```
def my_details( name, age ):  
    print("Name: ", name)  
    print("Age ", age)  
    return  
my_details(age=56,name="george")
```

- **Output:**

Name: george
Age 56

3. Default Arguments:

- Assumes a default value if a value is not provided in the function call for that argument.

```
def my_details( name, age=40 ):  
    print("Name: ", name)  
    print("Age ", age)  
    return  
my_details(name="george")
```

- **Output:**

Name: george

Age 40

4. Variable length Arguments

- If we want to specify more arguments than specified while defining the function, variable length arguments are used.
- It is denoted by *symbol before parameter.

```
def my_details(*name ):  
    print(*name)  
my_details("rajan","rahul","micheal",ärjun")
```

- **Output:**

rajan rahul micheal ärjun

Positional-Only Arguments

- You can specify that a function can have ONLY positional arguments, or ONLY keyword arguments.
- To specify that a function can have only positional arguments, add `, /` after the arguments:

Example

```
def my_function(x, /):
```

```
    print(x)
```

```
my_function(3)
```

```
def my_function(x, /):  
    print(x)
```

```
my_function(3)
```

```
def  
my_function(x):  
    print(x)
```

```
my_function(x = 3)
```

//error

```
def my_function(x, /):  
    print(x)
```

```
my_function(x = 3)
```

Recursive Function

A Python function can be called from within its body. When we do so it is called a recursive function.

```
def fun( ) :  
    # some statements  
    fun( ) # recursive call
```

Recursive call keeps calling the function again and again, leading to an infinite loop.

A provision must be made to get outside this infinite recursive loop. This is done by making the recursive call either in if block or in else block as shown below:

```
def fun( ) :  
    if condition :  
        # some statements  
    else  
        fun( ) # recursive call
```

```
def fun( ) :  
    if condition :  
        fun( )  
    else  
        # some statements
```

- Variables created in a function die when control returns from a function.
- Recursive function may or may not have a return statement.
- Typically, during execution of a recursive function many recursive calls happen, so several sets of variables get created. This increases the space requirement of the function.
- Recursive functions are inherently slow since passing value(s) and control to a function and returning value(s) and control will slow down the execution of the function.
- Recursive calls terminate when the base case condition is satisfied.

Recursive Factorial Function

```
def refact(n) :  
    if n == 0 :  
        return 1  
    else :  
        p = n * refact(n - 1)  
    return p  
  
num = int(input('Enter any number: '))  
fact = refact(num)  
print('Factorial value = ', fact)
```

Lambda Function

- A lambda function is a small anonymous function.
- A lambda function can take any number of arguments, but can only have one expression.

Python Lambda Functions are anonymous functions means that the function is without a name. As we already know the *def* keyword is used to define a normal function in Python. Similarly, the *lambda* keyword is used to define an anonymous function in [Python](#).

- lambda *arguments* : *expression*
- x = lambda a : a + 10
print(x(5))

- `def myfun(n):
 return lambda a : a * n`

- `def myfun(n):
 return lambda a : a * n`

`mydoubler = myfun(2)`

`print(mydoubler(11))`

- **MODULES:**

- A module is a file containing Python definitions, functions, statements and instructions.
- Standard library of Python is extended as modules.
- To use these modules in a program, programmer needs to import the module.
- Once we import a module, we can reference or use to any of its functions or variables in our code.
 - There is large number of standard modules also available in python.
 - Standard modules can be imported the same way as we import our user-defined modules.
 - Every module contains many function.
 - To access one of the function , you have to specify the name of the module and the name of the function separated by dot . This format is called dot notation.

- **Syntax:**

```
import module_name  
module_name.function_name(variable)
```

- **Importing Builtin Module:**

```
import math  
x=math.sqrt(25)  
print(x)
```

- **Importing User Defined Module:**

```
import cal  
x=cal.add(5,4)  
print(x)
```

There are four ways to import a module in our program, they are

Import: It is simplest and most common way to use modules in our code.

Example:

```
import math  
x=math.pi  
print("The value of pi is", x)
```

Output: The value of pi is 3.141592653589793

from import : It is used to get a specific function in the code instead of complete file.

Example:

```
from math import pi  
x=pi  
print("The value of pi is", x)
```

Output: The value of pi is 3.141592653589793

import with renaming:

We can import a module by renaming the module as our wish.

Example:

```
import math as m  
x=m.pi  
print("The value of pi is", x)
```

Output: The value of pi is 3.141592653589793

import all:

We can import all names(definitions) form a module using *

Example:

```
from math import *  
x=pi  
print("The value of pi is", x)
```

Output: The value of pi is 3.141592653589793

- Built-in python modules are,

1.math - mathematical functions:

- some of the functions in math module is,
- `math.ceil(x)` - Return the ceiling of x , the smallest integer greater than or equal to x
- `math.floor(x)` - Return the floor of x , the largest integer less than or equal to x .
- `math.factorial(x)` -Return x factorial.
- `math.gcd(x,y)`- Return the greatest common divisor of the integers a and b
- `math.sqrt(x)`- Return the square root of x
- `math.log(x)`- return the natural logarithm of x `math.log10(x)` - returns the base-10 logarithms `math.log2(x)` - Return the base-2 logarithm of x . `math.sin(x)` - returns sin of x radians
- `math.cos(x)`- returns cosine of x radians
- `math.tan(x)`-returns tangent of x radians
- `math.pi` - The mathematical constant $\pi = 3.141592$ `math.e` – returns The mathematical constant $e = 2.718281$

Thank You



NITTE
(Deemed to be University)

**NMAM INSTITUTE
OF TECHNOLOGY**

Nitte (DU) established under Section 3 of UGC Act 1956 | Accredited with 'A+' Grade by NAAC

Introduction To Python Programming

CS1001-2



Introduction to Python

Unit 2

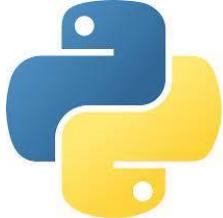
Data structures and Files

By

Prithviraj Jain
Dept. of CSE,
Nitte(Deemed to be University)

Python Data Structures





Python Strings

- String literal
- Multiline strings
- Accessing Elements
- Membership operator with strings
- Using + and * with strings
- Using Format()
- Escape Characters
- String methos

Python Strings – String literal

- String literals in python are surrounded by either single quotation marks, or double quotation marks.

'hello' is the same as "hello"

- Assigning a string to a variable is done with the variable name followed by an equal sign and the string:

```
a = "Hello"  
print(a)
```

Python Strings - multiline strings

- You can assign a multiline string to a variable by using three quotes (double or single):

Example-1

```
a = """Lorem ipsum dolor sit amet,  
consectetur adipiscing elit,  
sed do eiusmod tempor incididunt  
ut labore et dolore magna aliqua."""
```

```
print(a)
```

Note: in the result, the line breaks are inserted at the same position as in the code.

Python Strings - multiline strings

- Following example shows multiline string surrounded by triple single quotes

Example-2

```
a = "Lorem ipsum dolor sit amet,  
consectetur adipiscing elit,  
sed do eiusmod tempor incididunt  
ut labore et dolore magna aliqua"
```

```
print(a)
```

Python Strings - Accessing Elements

- Strings in Python are arrays of bytes representing unicode characters.
- However, Python does not have a character data type, a single character is simply a string with a length of 1
- Square brackets can be used to access elements of the string.

Example

```
a = "Hello, World!"  
print(a[1])
```

Output:
e

Python Strings - Accessing Elements

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

Example :Get the characters from position 2 to position 5 (not included):

```
b = "Hello, World!"  
print(b[2:5])
```

Output:
llo

Python Strings - Accessing Elements

- The term ‘string slice’ refers to a part of the string, where strings are sliced using a range of indices. For a string say name[n : m] where n and m are integers and legal indices. Python will return a slice of the string by returning the characters falling between indices n and m starting at n,n+1,n+2... till m-1.

Let's understand Slicing in strings with the help of few examples.

String word	a	m	a	z	i	n	g
Positive Index	0	1	2	3	4	5	6
Negative Index	-7	-6	-5	-4	-3	-2	-1

Python Strings - Accessing Elements

- **Slice From the Start**

Get the characters from the start to position 5 (not included):

```
b = "Hello, World!"  
print(b[:5])
```

Output:
Hello

Python Strings - Accessing Elements

- **Slice To the End**

Get the characters from position 2,
and all the way to the end:

```
b = "Hello, World!"  
print(b[2:])
```

Output:
llo, World!

Python Strings - Accessing Elements

- **Negative Indexing** : Use negative indexes to start the slice from the end of the string:
(Count from -1 from the end for negative index)

Example : Get the characters from position -5 to position -2 (not included)

```
b = "Hello, World!"  
print(b[-5:-2])
```

Output:
orl

Python Strings - Accessing Elements

- **More Example for string slicing :**

```
x = "Hello, Python"
```

```
print(x[-5:])
```

Output:
ython

```
print(x[:-3])
```

Output:
Hello, Pyt

```
print(x[7:-1])
```

Output:
Pytho

```
print(x[:])
```

Output:
Hello, Python

Python Strings - Accessing Elements

- You can also include a third number in slicing to indicate skipping step. In the following example, we start from index 1 and end with index 6 index(stops at $6-1=5$), and the skipping step is 2. It is a good example of Python slicing string by character.

```
x = "Hello, Python"  
print(x[1:6:2])
```

Output:
el,

Python Strings - Accessing Elements

- Also when skipping step is negative, string slicing is reversed. In this case first number indicates start index in the reverse order, second number is end index (not included) in the reverse order, and the last number is skipping step in the reverse order.
- Examples :

```
x = "Hello, Python"  
print(x[4:1:-1])
```

Output:
oll

Python Strings - Accessing Elements

Examples :

```
x = "Hello, Python"
```

Output:
,le

```
print(x[5:0:-2])
```

Output:
y olH

```
print(x[8: : -2])
```

Output:
nhy

```
print(x[: 6: -2])
```

Output:
nohtyP ,olleH

```
print(x[ : :-1])
```

Python Strings - Accessing Elements

- Examples :

```
x = "Hello, Python"
```

```
print(x[-2:7:-1])
```

Output:
ohty

```
print(x[4:-12:-1])
```

Output:
oll

```
print(x[-2:-6:-1])
```

Output:
ohty

Python Strings – Getting String Length

- To get the length of a string, use the `len()` function

Example :

```
a = "Hello, World!"  
print(len(a))
```

Output:
13

Finding minimum and maximum values

Example:

```
a = "Python"  
print(min(a))
```

Output:
P

```
print(min(a))
```

Output:
y

Python string - membership operator

- To check if a certain phrase or character is present in a string, we can use the keywords **in** or **not in**.

Example : Check if the phrase "ain" is present in the following text:

```
text = "The rain in Spain stays mainly  
       in the plain"
```

```
x = "ain" in text
```

```
print(x)
```

```
x = "ain" not in text
```

```
print(x)
```

Output:

True

False

Python Strings – using +

- To concatenate, or combine, two strings you can use the + operator.
- Example :

```
a = "Hello"  
b = "World"  
c = a + b  
print(c)
```

Output:
HelloWorld

Python Strings - using +

Example :

To add a space between them, add a "
":

```
a = "Hello"  
b = "World"  
c = a + " " + b  
print(c)
```

Output:
Hello World

Python Strings - using *

- (Replication) operator: The * operator replicates the string the number of times the value given on the other side.
- Consider the following example `>>>3*"go"`
`"gogogo"`

Python Strings - format()

- The format() method takes unlimited number of arguments, and are placed into the respective placeholders:

Example :

```
quantity = 3
itemno = 567
price = 49.95
myorder = "I want {} pieces of item {} for {} dollars."
print(myorder.format(quantity, itemno, price))
```

Output:

I want 3 pieces of item 567 for 49.95 dollars.

Python Strings - `format()`

- You can use index numbers inside {} to make sure that arguments are placed in the correct placeholders:

Example :

```
quantity = 3
itemno = 567
price = 49.95
myorder = "I want to pay {2} dollars for {0}
pieces of item {1}."
print(myorder.format(quantity, itemno, price))
```

Output:

I want to pay 49.95 dollars for 3 pieces of item 567

Python escape character

- To insert characters that are illegal in a string, use an escape character.
- An escape character is a backslash \ followed by the character you want to insert.

Example :

```
txt = "We are the so-called \"Vikings\" from the  
north."
```

```
Print(txt)
```

Output:

```
We are the so-called "Vikings" from the north.
```

Python escape character

- Other escape characters used in Python:

Code	Result
\'	Single Quote
\\"	Backslash
\n	New Line
\r	Carriage Return
\t	Tab
\b	Backspace
\f	Form Feed
\ooo	Octal value
\xhh	Hex value

Python String Formatting Operator

- One of Python's coolest features is the string format operator %.
- This operator is unique to strings and makes up for the lack of having functions from C's printf() family.
Following is a simple example –

```
print "My name is %s and weight is %d kg!" % ('Zara', 21)
```

Output : My name is Zara and weight is 21 kg!

Python String Formatting Operator

- Here is the list of complete set of symbols which can be used along with % -

Format Symbol	Conversion
%c	character
%s	string conversion via str() prior to formatting
%i	signed decimal integer
%d	signed decimal integer
%u	unsigned decimal integer
%o	octal integer
%x	hexadecimal integer (lowercase letters)
%X	hexadecimal integer (UPPERcase letters)

Python String - methods

- The `strip()` method removes any whitespace from the beginning or the end:

Example :

```
a = "Hello, World! "
print(a.strip()) # returns "Hello, World!"
```

Python String - methods

- The lower() and upper() method returns the string in lower and upper case respectively:

Example :

```
a = "Hello, World!"  
print(a.lower())  
print(a.upper())
```

Output:
hello, world!
HELLO, WORLD!

Python String - methods

- The replace() method replaces a string with another string:

Example :

```
a = "Hello, World!"  
print(a.replace("H", "J"))
```

Output:
Jello, World!

Python String - methods

- The `split()` method splits the string into substrings if it finds instances of the separator:

Example :

```
a = "Hello, World!"  
print(a.split(","))
```

Output:

```
['Hello', ' World!']
```

Python String - All methods

Method	Description
<u>capitalize()</u>	Converts the first character to upper case
<u>casefold()</u>	Converts string into lower case
<u>center()</u>	Returns a centered string
<u>count()</u>	Returns the number of times a specified value occurs in a string
<u>encode()</u>	Returns an encoded version of the string
<u>endswith()</u>	Returns true if the string ends with the specified value

Python String - All methods

expandtabs() Sets the tab size of the string

find() Searches the string for a specified value and returns the position of where it was found

format() Formats specified values in a string

format_map() Formats specified values in a string

index() Searches the string for a specified value and returns the position of where it was found

isalnum() Returns True if all characters in the string are alphanumeric

isalpha() Returns True if all characters in the string are in the alphabet

isascii() Returns True if all characters in the string are ascii characters

Python String - All methods

[isdecimal\(\)](#) Returns True if all characters in the string are decimals

[isdigit\(\)](#) Returns True if all characters in the string are digits

[isidentifier\(\)](#) Returns True if the string is an identifier

[islower\(\)](#) Returns True if all characters in the string are lower case

[isnumeric\(\)](#) Returns True if all characters in the string are numeric

[isprintable\(\)](#) Returns True if all characters in the string are printable

[isspace\(\)](#) Returns True if all characters in the string are whitespaces

[istitle\(\)](#) Returns True if the string follows the rules of a title

Python String - All methods

isupper() Returns True if all characters in the string are upper case

join() Converts the elements of an iterable into a string

ljust() Returns a left justified version of the string

lower() Converts a string into lower case

lstrip() Returns a left trim version of the string

maketrans() Returns a translation table to be used in translations

partition() Returns a tuple where the string is parted into three parts

Python String - All methods

rindex() Searches the string for a specified value and returns the last position of where it was found

rjust() Returns a right justified version of the string

rpartition() Returns a tuple where the string is parted into three parts

rsplit() Splits the string at the specified separator, and returns a list

rstrip() Returns a right trim version of the string

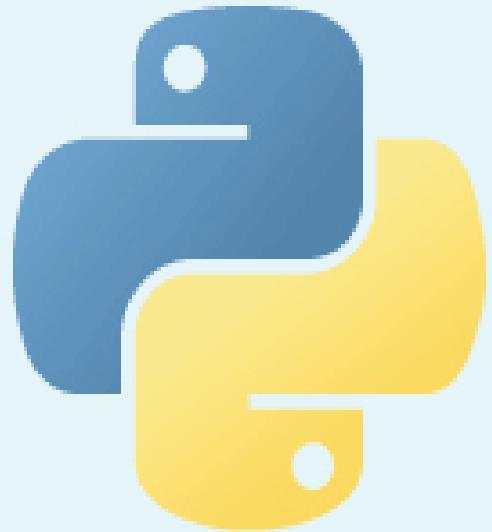
split() Splits the string at the specified separator, and returns a list

splitlines() Splits the string at line breaks and returns a list

startswith() Returns true if the string starts with the specified value

Python String - All methods

<u>strip()</u>	Returns a trimmed version of the string
<u>swapcase()</u>	Swaps cases, lower case becomes upper case and vice versa
<u>title()</u>	Converts the first character of each word to upper case
<u>translate()</u>	Returns a translated string
<u>upper()</u>	Converts a string into upper case
<u>zfill()</u>	Fills the string with a specified number of 0 values at the beginning



["Python", 11, "C", "R"]

Python Lists

Python Lists

- A list is a collection which is ordered and changeable.
In Python lists are written with square brackets.
- We can store all types of items (including another list) in a list. A list may contain mixed type of items

Example:

```
thislist = ["apple", "banana", "cherry"]  
print(thislist)
```

Output:

```
['apple', 'banana', 'cherry']
```

Python Lists

- Example: Print the second item of the list:

```
thislist = ["apple", "banana", "cherry"]  
print(thislist[1])
```

Output:
banana

- Negative indexing means beginning from the end, -1 refers to the last item, -2 refers to the second last item etc.

```
thislist = ["apple", "banana", "cherry"]  
print(thislist[-1])
```

Output:
cherry

Python Lists

You can specify a range of indexes by specifying where to start and where to end the range.

When specifying a range, the return value will be a new list with the specified items.

Example :

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi",
"melon", "mango"]
print(thislist[2:5])
```

Output : ['cherry', 'orange', 'kiwi']

The search will start at index 2 (included) and end at index 5 (not included).

Python Lists

Example :This example returns the items from the beginning to "orange"

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon",
            "mango"]
print(thislist[:4])
```

#output : ['apple', 'banana', 'cherry', 'orange']

Example :This example returns the items from "cherry" and to the end:

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon",
            "mango"]
print(thislist[2:])
```

#output : ['cherry', 'orange', 'kiwi', 'melon', 'mango']

Python Lists

- Specify negative indexes if you want to start the search from the end of the list:

Example : This example returns the items from index -4 (included) to index -1 (excluded)

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon",
           "mango"]
print(thislist[-4:-1])
```

#output : ['orange', 'kiwi', 'melon']

Python Lists

To change the value of a specific item, refer to the index number:

Example : Change the second item:

```
thislist = ["apple", "banana", "cherry"]  
thislist[1] = "blackcurrant"  
print(thislist)
```

#output : ['apple', 'blackcurrant', 'cherry']

Python Lists

- Print all items in the list, one by one:

Example :

```
thislist = ["apple", "banana", "cherry"]
for x in thislist:
    print(x)
```

#output : apple
banana
cherry

Python Lists

- Check if "apple" is present in the list::

Example :

```
thislist = ["apple", "banana", "cherry"]
if "apple" in thislist:
    print("Yes, 'apple' is in the fruits list")
```

#output : Yes, 'apple' is in the fruits list

Python Lists

- Print the number of items in the list:

Example :

```
thislist = ["apple", "banana", "cherry"]  
print(len(thislist))
```

#output : 3

Python Lists

- To add an item to the end of the list, use the append() method:
- Example :

```
thislist = ["apple", "banana", "cherry"]  
thislist.append("orange")  
print(thislist)
```

#output : ['apple', 'banana', 'cherry', 'orange']

Python Lists

- To add an item at the specified index, use the `insert()` method:
- Example :

```
thislist = ["apple", "banana", "cherry"]
thislist.insert(1, "orange")
print(thislist)
```

#output : ['apple', 'orange', 'banana', 'cherry']

Python Lists

- The remove() method removes the specified item:
- Example :

```
thislist = ["apple", "banana", "cherry"]
thislist.remove("banana")
print(thislist)
```

#output : ['apple', 'cherry']

Python Lists

- The pop() method removes the specified index, (or the last item if index is not specified):
- Example :

```
thislist = ["apple", "banana", "cherry"]  
thislist.pop()  
print(thislist)
```

#output : ['apple', 'banana']

```
thislist = ["apple", "banana", "cherry"]
```

```
    thislist.pop(1) #remove let at 1  
    print(thislist)
```

#output : ['apple', "cherry"]

Python Lists

- The del keyword removes the specified index:
- Example :

```
thislist = ["apple", "banana", "cherry"]
del thislist[0]
print(thislist)
```

#output : ['banana', 'cherry']

Python Lists

- The del keyword can also delete the list completely:
- Example :

```
thislist = ["apple", "banana", "cherry"]
del thislist
print(thislist)
```

#output : Error

Python Lists

- The clear() method empties the list:
- Example :

```
thislist = ["apple", "banana", "cherry"]
thislist.clear()
print(thislist)
```

#output : []

Python Lists

- Make a copy of a list with the `copy()` method:
- Example :

```
thislist = ["apple", "banana", "cherry"]
mylist = thislist.copy()
print(mylist)
```

#output : ['apple', 'banana', 'cherry']

Python Lists

- Make a copy of a list with the list() method:
- Example :

```
thislist = ["apple", "banana", "cherry"]
mylist = list(thislist)
print(mylist)

#output : ['apple', 'banana', 'cherry']
```

Python Lists

- Join two list with +
- Example :

```
list1 = ["a", "b" , "c"]  
list2 = [1, 2, 3]  
list3 = list1 + list2  
print(list3)
```

#output : ['a', 'b', 'c', 1, 2, 3]

Python Lists

- Append list2 into list1: (same as +)
- Example :

```
list1 = ["a", "b", "c"]  
list2 = [1, 2, 3]
```

```
for x in list2:  
    list1.append(x)
```

```
print(list1)  
#output : ['a', 'b', 'c', 1, 2, 3]
```

Python Lists

- Use the `extend()` method to add list2 at the end of list1:
(same as `append`)
- Example :

```
list1 = ["a", "b", "c"]
list2 = [1, 2, 3]
list1.extend(list2)
print(list1)
```

#output : ['a', 'b', 'c', 1, 2, 3]

Python Lists

- It is also possible to use the list() constructor to make a new list.
- Example :

```
thislist = list(("apple", "banana", "cherry")) # note  
the double round-brackets  
print(thislist)
```

#output : ['apple', 'banana', 'cherry']

Python Lists

- Sorting the list

Syntax : Syntax

list.sort(reverse=True|False, key=myFunc)

reverse (Optional) : reverse=True will sort the list descending. Default is reverse=False

key (optional) : A function to specify the sorting criteria(s)

Python Lists

- Use sort() to sort the list
- Example :

```
mylist = ["banana", "cherry", "apple"]
mylist.sort()
print(mylist)
```

#output : ['apple', 'banana', 'cherry']

Python Lists

- Use `sort(reverse=True)` to sort the list in reverse order.
- Example :

```
cars = ['Ford', 'BMW', 'Volvo']
cars.sort(reverse=True)
Print(cars)
```

```
#output : ['Volvo', 'Ford', 'BMW']
```

Python Lists

- You can Sort the list by the length of the values:

Example :

```
# A function that returns the length of the value:  
def myFun(e):  
    return len(e)
```

```
cars = ['Ford', 'Mitsubishi', 'BMW', 'VW']
```

```
cars.sort(key=myFun)
```

Python Lists

- Python has a set of built-in methods that you can use on lists.

Method	Description
<u>append()</u>	Adds an element at the end of the list
<u>clear()</u>	Removes all the elements from the list
<u>copy()</u>	Returns a copy of the list
<u>count()</u>	Returns the number of elements with the specified value
<u>extend()</u>	Add the elements of a list (or any iterable), to the end of the current list
<u>index()</u>	Returns the index of the first element with the specified value
<u>insert()</u>	Adds an element at the specified position

Python Lists

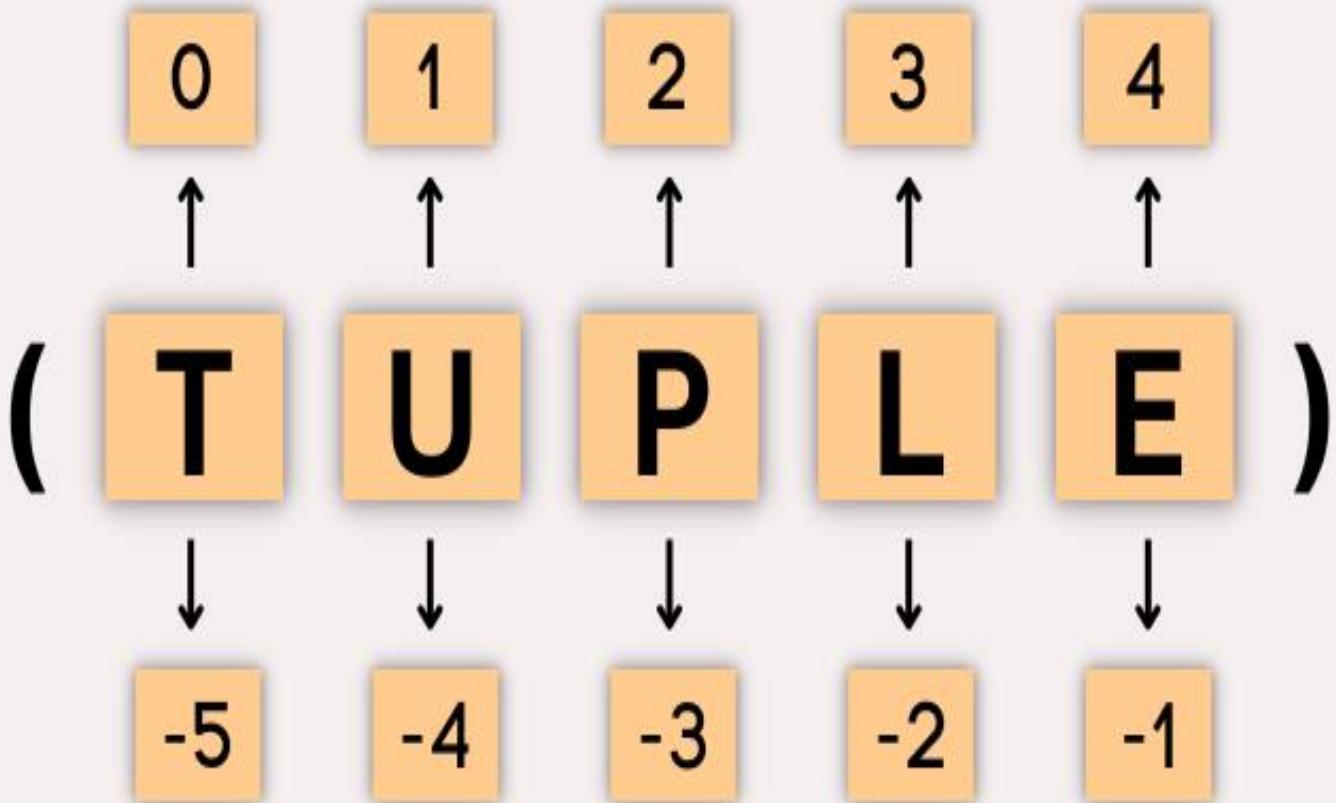
pop() Removes the element at the specified position

remove() Removes the item with the specified value

reverse() Reverses the order of the list

sort() Sorts the list

Tuples , Sets, Dictionaries



- A tuple is same as list, except that the set of elements is enclosed **in parentheses** instead of square brackets.
- A tuple is an **immutable** list. i.e. once a tuple has been created, you can't add elements to a tuple or remove elements from the tuple.
- Tuple items are ordered, unchangeable, and allow duplicate values.
- But tuple can be converted into list and list can be converted into tuple.

methods	example	description
list()	<pre>>>> a=(1,2,3,4,5) >>> a=list(a) >>> print(a) [1, 2, 3, 4, 5]</pre>	it convert the given tuple into list.
tuple()	<pre>>>> a=[1,2,3,4,5] >>> a=tuple(a) >>> print(a) (1, 2, 3, 4, 5)</pre>	it convert the given list into tuple.

Benefit of Tuple:

- Tuples are faster than lists.
- If the user wants to protect the data from accidental changes, tuple can be used.
- Tuples can be used as keys in dictionaries, while lists can't.

Operations on Tuples:

1. Indexing
2. Slicing
3. Concatenation
4. Repetitions
5. Membership
6. Comparison

Operations	examples	description
Creating a tuple	<code>>>>a=(20,40,60,"apple","ball")</code>	Creating the tuple with elements of different data types.
Indexing	<code>>>>print(a[0]) 20 >>>a[2] 60</code>	Accessing the item in the position 0 Accessing the item in the position 2
Slicing	<code>>>>print(a[1:3]) (40,60)</code>	Displaying items from 1st till 2nd.
Concatenation	<code>>>>b=(2,4) >>>print(a+b) >>>(20,40,60,"apple","ball",2,4)</code>	Adding tuple elements at the end of another tuple elements
Repetition	<code>>>>print(b*2) >>>(2,4,2,4)</code>	repeating the tuple in n no of times
Membership	<code>>>>a=(2,3,4,5,6,7,8,9,10) >>>5 in a True >>>100 in a False >>>2 not in a False</code>	Returns True if element is present in tuple. Otherwise returns false.
Comparison	<code>>>>a=(2,3,4,5,6,7,8,9,10) >>>b=(2,3,4) >>>a==b False >>>a!=b True</code>	Returns True if all elements in both elements are same. Otherwise returns false

Tuple methods:

- Tuple is immutable so changes cannot be done on the elements of a tuple once it is assigned.

methods	example	description
a.index(tuple)	>>> a=(1,2,3,4,5) >>> a.index(5) 4	Returns the index of the first matched item.
a.count(tuple)	>>>a=(1,2,3,4,5) >>>a.count(3) 1	Returns the count of the given element.
len(tuple)	>>> len(a) 5	return the length of the tuple
min(tuple)	>>> min(a) 1	return the minimum element in a tuple
max(tuple)	>>> max(a) 5	return the maximum element in a tuple
del(tuple)	>>> del(a)	Delete the entire tuple.

Tuple Assignment:

- Tuple assignment allows variables on the left of an assignment operator and values of tuple on the right of the assignment operator.
- Multiple assignment works by creating a tuple of expressions from the right hand side, and a tuple of targets from the left, and then matching each expression to a target.
- Because multiple assignments use tuples to work, it is often termed tuple assignment.

Uses of Tuple assignment:

- It is often useful to swap the values of two variables.
- Example:
- Swapping using temporary variable:

a=20

b=50

temp = a

a = b

b = temp

print("value after swapping is",a,b)

Swapping using tuple assignment:

a=20

b=50

(a,b)=(b,a)

print("value after swapping is",a,b)

Multiple assignments:

- Multiple values can be assigned to multiple variables using tuple assignment.

```
>>> (a,b,c)=(1,2,3)
>>> print(a)
1
>>> print(b)
2
>>> print(c)
3
```

Tuple as return value:

- A Tuple is a comma separated sequence of items.
- It is created with or without ().
- A function can return one value. if you want to return more than one value from a function. we can use tuple as return value.

Eg 1

```
def div(a,b):  
    r=a%b  
    q=a//b  
    return(r,q)  
  
a=int(input("enter a value:"))  
b=int(input("enter b value:"))  
r,q=div(a,b)  
print("remainder:",r)  
print("quotient:",q)
```

Output:

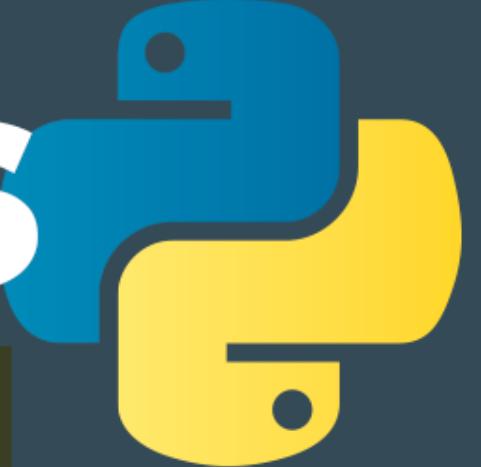
```
enter a value:4  
enter b value:3  
remainder: 1  
quotient: 1
```

Eg:2

```
def minmax(a):  
    small=min(a)  
    big=max(a)  
    return(small, big)  
  
a=[1,2,3,4,6]  
small, big=minmax(a)  
print("smallest:", small)  
print("biggest:", big)
```

Output:
smallest: 1
biggest: 6

Sets



{1, 2, 3, 4}

A Set in Python is used to store a collection of items with the following properties.

- 1) No duplicate elements. If try to insert the same item again, it overwrites previous one.
- 2) An unordered collection. When we access all items, they are accessed without any specific order and we cannot access items using indexes as we do in lists.

Unchangeable:

Set items are unchangeable, meaning that we cannot change the items after the set has been created.

```
s = {10, 50, 20}
```

```
print(s)
```

```
print(type(s))
```

Output

```
{10, 50, 20}
```

```
<class 'set'>
```

Type Casting with Python Set method

The Python set() method is used for type casting.



```
1 # typecasting list to set  
2 s = set(["a", "b", "c"])  
3 print(s)  
  
4  
5 # Adding element to the set  
6 s.add("d")  
7 print(s)
```



Output

```
{'c', 'b', 'a'}  
{'d', 'c', 'b', 'a'}
```

Accessing Set Elements

```
thisset = {"apple", "banana", "cherry"}
```

```
for x in thisset:  
    print(x)
```

Python - Add Set Items

- Add an item to a set, using the add() method:

Add an item to a set, using the `add()` method:

```
thisset = {"apple", "banana", "cherry"}  
thisset.add("orange")  
  
print(thisset)
```

Python - Remove Set Items

Remove Item

To remove an item in a set, use the `remove()`, or the `discard()` method.(If the item to remove does not exist, `remove()` will raise an error)

Example

Remove "banana" by using the `remove()` method:

```
thisset = {"apple", "banana", "cherry"}  
  
thisset.remove("banana")  
  
print(thisset)
```

- Remove "banana" by using the discard() method: If the item to remove does not exist, discard() will NOT raise an error.

Remove banana by using the discard() method.

```
thisset = {"apple", "banana", "cherry"}  
thisset.discard("banana")  
print(thisset)
```

- You can also use the `pop()` method to remove an item, but this method will remove a random item, so you cannot be sure what item that gets removed.
- The return value of the `pop()` method is the removed item.

Remove a random item by using the `pop()` method:

```
thisset = {"apple", "banana", "cherry"}  
  
x = thisset.pop()  
  
print(x)  
  
print(thisset)
```

The `del` keyword will delete the set completely.

The `clear()` method empties the set:

`thisset.clear()`

The `del` keyword will delete the set completely:

```
thisset = {"apple", "banana", "cherry"}
```

```
del thisset
```

```
print(thisset)
```

Join Sets

- There are several ways to join two or more sets in Python.
1. **union()** and **update()** : joins all items from both sets.
 2. **intersection()** : keeps ONLY the duplicates.
 3. **difference()** : keeps the items from the first set that are not in the other set(s).
 4. **symmetric_difference()** : method keeps all items EXCEPT the duplicates.

```
x = {"apple", "banana", "cherry"}  
y = {"google", "microsoft", "apple"}
```

```
z = x.union(y)
```

```
print(z)
```

Output:

{'google', 'banana', 'microsoft', 'apple', 'cherry'}

Use | as a shortcut instead of union():

```
x = {"apple", "banana", "cherry"}  
y = {"google", "microsoft", "apple"}
```

```
z = x | y
```

```
print(z)
```

Output:

```
{'apple', 'cherry', 'microsoft', 'google', 'banana'}
```

Add Sets

Example

Add elements from `tropical` into `thisset`:

```
thisset = {"apple", "banana", "cherry"}  
tropical = {"pineapple", "mango", "papaya"}  
  
thisset.update(tropical)  
  
print(thisset)
```

Add Any Iterable

- The object in the update() method does not have to be a set, it can be any iterable object (tuples, lists, dictionaries etc.).

Example

Add elements of a list to at set:

```
thisset = {"apple", "banana", "cherry"}  
mylist = ["kiwi", "orange"]  
  
thisset.update(mylist)  
  
print(thisset)
```

```
x = {"apple", "banana", "cherry"}  
y = {"google", "microsoft", "apple"}
```

```
z = x.intersection(y)
```

```
print(z)
```

keeps ONLY the duplicates

```
x = {"apple", "banana", "cherry"}  
y = {"google", "microsoft", "apple"}
```

```
z = x & y
```

```
print(z)
```

Output:
{'apple'}

```
x = {"apple", "banana", "cherry"}
```

```
y = {"google", "microsoft", "apple"}
```

```
z = x.difference(y)
```

```
print(z)
```

Output:
{'banana', 'cherry'}

**keeps the items from the first set that are not in
the other set(s).**

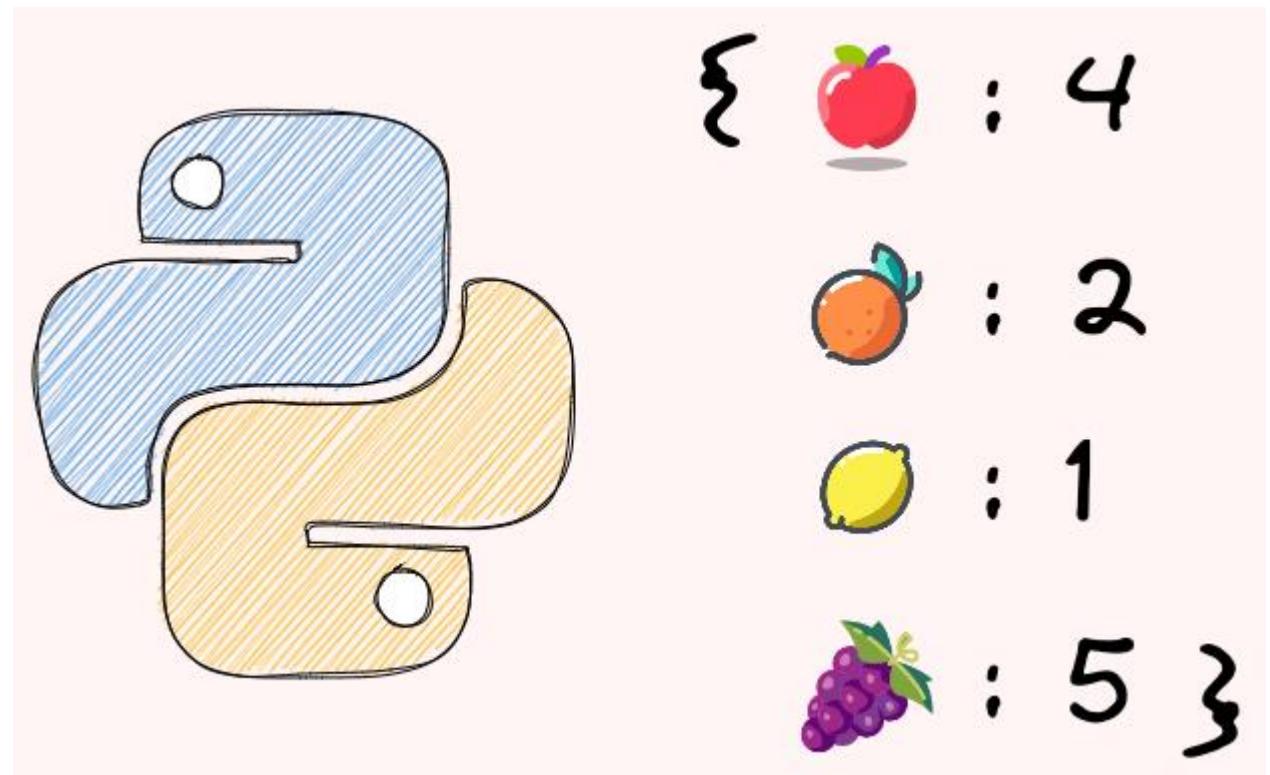
Return a set that contains all items from both sets, except items that are present in both sets:

- `x = {"apple", "banana", "cherry"}`
- `y = {"google", "microsoft", "apple"}`
- `z = x.symmetric_difference(y)`
- `print(z)` method keeps all items EXCEPT the duplicates.

Output:

`{'cherry', 'google', 'banana', 'microsoft'}`

Python Dictionary



Dictionaries

- Dictionary is an unordered collection of elements.
- An element in dictionary has a **key: value** pair.
- All elements in dictionary are placed inside the curly braces i.e. { }
- Elements in Dictionaries are **accessed via keys** and not by their position.
- The values of a dictionary can be any data type. Duplicate items are not allowed.
- Keys must be immutable data type (numbers, strings, tuple)

Operations on dictionary:

1. Accessing an element
2. Update
3. Add element
4. Membership

Operations	Example	Description
Creating a dictionary	<code>>>> a={1:"one",2:"two"} >>> print(a) {1: 'one', 2: 'two'}</code>	Creating the dictionary with elements of different data types.
accessing an element	<code>>>> a[1] 'one' >>> a[0] KeyError: 0</code>	Accessing the elements by using keys.
Update	<code>>>> a[1]="ONE" >>> print(a) {1: 'ONE', 2: 'two'}</code>	Assigning a new value to key. It replaces the old value by new value.
add element	<code>>>> a[3]="three" >>> print(a) {1: 'ONE', 2: 'two', 3: 'three'}</code>	Add new element in to the dictionary with key.
membership	<code>a={1: 'ONE', 2: 'two', 3: 'three'} >>> 1 in a True >>> 3 not in a False</code>	Returns True if the key is present in dictionary. Otherwise returns false.

a.update(dictionary)	<pre>>>> b={4:"four"} >>> a.update(b) >>> print(a) {1: 'ONE', 2: 'two', 3: 'three', 4: 'four'}</pre>	It will add the dictionary with the existing dictionary
fromkeys()	<pre>>>> key={"apple","ball"} >>> value="for kids" >>> d=dict.fromkeys(key,value) >>> print(d) {'apple': 'for kids', 'ball': 'for kids'}</pre>	It creates a dictionary from key and values.
len(a)	<pre>a={1: 'ONE', 2: 'two', 3: 'three'} >>> len(a) 3</pre>	It returns the length of the list.
clear()	<pre>a={1: 'ONE', 2: 'two', 3: 'three'} >>> a.clear() >>> print(a) >>> {}</pre>	Remove all elements form the dictionary.
del(a)	<pre>a={1: 'ONE', 2: 'two', 3: 'three'} >>> del(a)</pre>	It will delete the entire dictionary.

Methods in dictionary:

Method	Example	Description
a.copy()	a={1: 'ONE', 2: 'two', 3: 'three'} >>> b=a.copy() >>> print(b) {1: 'ONE', 2: 'two', 3: 'three'}	It returns copy of the dictionary. here copy of dictionary 'a' get stored in to dictionary 'b'
a.items()	>>> a.items() dict_items([(1, 'ONE'), (2, 'two'), (3, 'three')])	Return a new view of the dictionary's items. It displays a list of dictionary's (key, value) tuple pairs.
a.keys()	>>> a.keys() dict_keys([1, 2, 3])	It displays list of keys in a dictionary
a.values()	>>> a.values() dict_values(['ONE', 'two', 'three'])	It displays list of values in dictionary
a.pop(key)	>>> a.pop(3) 'three' >>> print(a) {1: 'ONE', 2: 'two'}	Remove the element with key and return its value from the dictionary.
setdefault(key,value)	>>> a.setdefault(3,"three") 'three' >>> print(a) {1: 'ONE', 2: 'two', 3: 'three'} >>> a.setdefault(2) 'two'	If key is in the dictionary, return its value. If key is not present, insert key with a value of dictionary and return dictionary.

Difference between List, Tuples and dictionary:

List	Tuples	Dictionary
A list is mutable Lists are dynamic	A tuple is immutable Tuples are fixed size in nature	A dictionary is mutable In values can be of any data type and can repeat, keys must be of immutable type
List are enclosed in brackets[] and their elements and size can be changed	Tuples are enclosed in parenthesis () and cannot be updated	Tuples are enclosed in curly braces {} and consist of key:value
<u>Example:</u> List = [10, 12, 15]	<u>Example:</u> Words = ("spam", "egss") Or Words = "spam", "eggs"	<u>Example:</u> Dict = {"ram": 26, "abi": 24}
<u>Access:</u> print(list[0])	<u>Access:</u> print(words[0])	<u>Access:</u> print(dict["ram"])
Can contain duplicate elements	Can contain duplicate elements. Faster compared to lists	Can't contain duplicate keys, but can contain duplicate values
Slicing can be done <u>Usage:</u> <ul style="list-style-type: none">❖ List is used if a collection of data that doesn't need random access.❖ List is used when data can be modified frequently	Slicing can be done <u>Usage:</u> <ul style="list-style-type: none">❖ Tuple can be used when data cannot be changed.❖ A tuple is used in combination with a dictionary i.e. a tuple might represent a key.	Slicing can't be done <u>Usage:</u> <ul style="list-style-type: none">❖ Dictionary is used when a logical association between key:value pair.❖ When in need of fast lookup for data, based on a custom key.❖ Dictionary is used when data is being constantly modified.

CLASSES & OBJECTS



python

OOP, Defining a Class

- Like other general-purpose programming languages, Python is also an object-oriented language since its beginning.
- It allows us to develop applications using an Object-Oriented approach.
- Object: The object is an entity that has state and behavior.
- Class: The class can be defined as a collection of objects.

Car Class



Maruti



Audi



BMW

Objects

Create a Class

To create a class, use the keyword `class`

Declaring a class:

```
class name:  
    statements
```

Ex:

```
class MyClass:
```

```
    x = 7.4
```

Create Object

Now we can use the class named MyClass to create objects:

class MyClass:

x=7.4

z=MyClass()

print(z.x)

Fields

name = value

- Example:

```
class Point:  
    x = 0  
    y = 0
```

```
# main  
p1 = Point()  
p1.x = 2  
p1.y = -5  
print(p1.x)
```

point.py

```
1 class Point:  
2     x = 0  
3     y = 0
```

can be declared directly inside class (as shown here)
or in constructors (more common)

Using a Class

```
import class
```

- client programs must import the classes they use

point_main.py

```
1 from Point import *
2
3 # main
4 p1 = Point()
5 p1.x = 7
6 p1.y = -3
7 ...
8
9 # Python objects are dynamic (can add fields any time!)
10 p1.name = "Tyler Durden"
```

Object Methods

```
def name(self, parameter, ..., parameter):  
    statements
```

- `self` *must* be the first parameter to any object method. It is a reference to the current instance of the class
- *must* access the object's fields through the `self` reference

```
class Point:  
    def translate(self, dx, dy):  
        self.x += dx  
        self.y += dy  
    ...
```

Write `distance`, `set_location`, and `distance_from_origin` **methods**.

point.py

```
1 from math import *
2
3 class Point:
4     x = 0
5     y = 0
6
7     def set_location(self, x, y):
8         self.x = x
9         self.y = y
10
11    def distance_from_origin(self):
12        return sqrt(self.x * self.x + self.y * self.y)
13
14    def distance(self, other):
15        dx = self.x - other.x
16        dy = self.y - other.y
17        return sqrt(dx * dx + dy * dy)
```

Calling Methods

- A client can call the methods of an object in two ways:
 - (the value of `self` can be an implicit or explicit parameter)

1) **object.method(parameters)**

or

2) **Class.method(object, parameters)**

- Example:

```
p = Point(3, -4)
p.translate(1, 5)
Point.translate(p, 1, 5)
```

The `__init__()` Function

- All classes have a function called `__init__()`, which is always executed when the class is being initiated.
- Use the `__init__()` function to assign values to object properties, or other operations that are necessary to do when the object is being created.

Syntax:

```
def __init__(self, parameter, ..., parameter) :  
    statements
```

`self` is a reference to the current instance of the class.
It is used to access variables and methods associated with a specific object

```
class Person:
```

```
    def __init__(self, name, age):  
        self.name = name  
        self.age = age
```

```
p1 = Person("John", 36)
```

```
print(p1.name)  
print(p1.age)
```

`str`

The `__str__` method in Python is a **special (magic/dunder) method** used to define how an object should be represented as a **string** when passed to functions

```
def __str__(self):  
    return string
```

- converts object to a string
- invoked automatically when `str` or `print` is called

Exercise: Write a `__str__` method for `Point` objects that returns strings like "`(3, -14)`"

```
def __str__(self):  
    return "(" + str(self.x) + ", " + str(self.y) + ")"
```

Complete Point Class

point.py

```
1  from math import *
2
3  class Point:
4      def __init__(self, x, y):
5          self.x = x
6          self.y = y
7
8      def distance_from_origin(self):
9          return sqrt(self.x * self.x + self.y * self.y)
10
11     def distance(self, other):
12         dx = self.x - other.x
13         dy = self.y - other.y
14         return sqrt(dx * dx + dy * dy)
15
16     def translate(self, dx, dy):
17         self.x += dx
18         self.y += dy
19
20     def __str__(self):
21         return "(" + str(self.x) + ", " + str(self.y) + ")"
```

INTRODUCTION

- ❖ **FILE HANDLING** is a mechanism by which we can **read data** of disk files in python program or **write back data** from python program to disk files.
- ❖ So far in python program the standard **input** was from **keyboard** an **output** was going to monitor i.e. no **data is stored permanently** and **entered data is present as long as program is running**.
- ❖ But **file handling** allows us to **store data** entered through python program permanently in disk file and later on we can read back the data.

Files

- Files are used to store data

A collection of data or information that are stored on computer known as **file**.

- Different types of files : Text, image, video, audio, binary, csv files
- Files are named location on memory (disk) to store related information
- Since RAM is volatile, there is a need to store data in files for future use.

- Files are storage unit on the hard disk.
- Files consist of two things:
 - i) File name ii) Extension
- Both are separated by dot
- Ex: program1.py
- abc.txt

- File handling in Python allows you to read from and write to files on your computer.
- Python has built-in functions to interact with different types of files, such as text files and CSV (Comma-Separated Values) files.

File Types:

- **Text files:** These files contain human-readable text.
- **CSV files:** These are text files where data is stored in a tabular format, separated by commas, which is used for storing structured data like spreadsheets.

File handling:

◎ We can:

- Open a file
- Write to a file
- Read from a file
- Close a file



File operations:

- When we want to read from or write to a file we need to open it first.
- When we are done, it needs to be closed, so that resources that are tied with the file are freed.
- Hence, in Python, a file operation takes place in the following order.
 - **Open file**
 - **Read or Write**
 - **Close file**

Text Files

- A text file is a file that contains printable characters and whitespace, organized in to lines separated by newline characters.
- Ex: .txt, .docx

Working with Text Files

Creating a Text File

To create a new text file, you can use the `open()` function in **write** mode ('w'). If the file doesn't exist, it will be created.

Reading from a Text File

To read the content of a text file, you use the `open()` function in **read** mode ('r').

Writing to a Text File

To write data into a text file, open it in **write** ('w') or **append** ('a') mode. Writing to a file in 'w' mode overwrites the existing content, while 'a' mode adds to the existing content.

Reading and Writing Files

- We can read a file or write to a file only after creating the file.
- The file must exist in order to perform any operations on it.
- Thus python has file operations to open, read, write and close the file

Opening a file

- ◎ Python has a built-in function **open()** to open a file.
- ◎ This function returns a file object, also called a handle, as it is used to read or modify the file accordingly.
- ◎ Syntax:

f=open(<filename>,<mode>)

Ex: f=open("file.txt")

This opens a file in current directory (working environment). Default mode is **read**.

Opening a file

- ◎ We can specify full pathname of the file (location where the file is present):

```
f=open('C:/Python3/README.txt')
```

- ◎ File can be opened in any modes: read, write append etc.

Opening a file : file modes

Character	Meaning
'r'	open for reading (default)
'w'	open for writing, truncating the file first
'x'	open for exclusive creation, failing if the file already exists
'a'	open for writing, appending to the end of the file if it exists
'b'	binary mode
't'	text mode (default)
'+'	open a disk file for updating (reading and writing)

Opening a file: Example

1. *f = open("test.txt") # equivalent to 'r' or 'rt'*
2. *f = open("test.txt",'w') # write in text mode*
3. *f =open("img.png",'r+b') # read and write in binary mode*

Closing a file

- ◎ When we are done with operations to the file, we need to properly close it.
- ◎ Closing a file will free up the resources that were tied with the file
 - done using the **close()** method.
- ◎ Python has a garbage collector to clean up unreferenced objects but, we must not rely on it to close the file.

Closing a file

Ex:

```
f=open("file.txt")
```

```
# perform file operations like writing to file or  
reading from the file
```

```
f.close()
```

Create a Text File and Write to It

```
# Open a file in write mode, this will create the file if it  
doesn't exist
```

```
file = open('example.txt', 'w')
```

```
# Write some text to the file
```

```
file.write("Hello, this is a text file.\n")
```

```
file.write("This is the second line.")
```

```
# Close the file to save changes
```

```
file.close()
```

Read from a Text File

- # Open the file in read mode

- file = open('example.txt', 'r')

- # Read the content of the file

- content = file.read()

- # Print the content of the file

- print(content)

- # Close the file after reading

- file.close()

Reading and Writing

- ◎ **REMEMBER:**
- ◎ open function opens an existing file in read mode by default.
- ◎ Opening a non existing file in write mode will create a new file.
- ◎ Opening an existing file in write mode will clear the content of the file.

Append to a Text File (Add More Data)

- # Open the file in append mode to add more content
- file = open('example.txt', 'a')
- # Add a new line at the end of the file
- file.write("\nThis is a new line added to the file.")
- # Close the file after appending
- file.close()

Example: readline()

Sample File



File Edit Format View Help

Python Programming Elective.
Department of Civil Engg.
NMAM Institute of Technology, Nitte.
Karkala-574110

Program

```
#readline( )
myfile=open("example.txt", "r")
line1 = myfile.readline()
line2 = myfile.readline()
line3 = myfile.readline(4)
print(line1, end="")
print(line2, end="")
print(line3)
```

Output

Python Programming Elective.
Department of Civil Engg.
NMAM

Example: readlines()

Sample File



File Edit Format View Help

Python Programming Elective.

Department of Civil Engg.

NMAM Institute of Technology, Nitte.

Karkala-574110

Program

```
#readlines( )
myfile=open("example.txt", "r")
line1 = myfile.readlines()
print(line1)
```

Output

```
['Python Programming Elective.\n', 'Department of Civil Engg.\n',
'NMAM Institute of Technology, Nitte.\n', 'Karkala-574110']
```

Example: readlines()

Sample File



File Edit Format View Help

Python Programming Elective.

Department of Civil Engg.

NMAM Institute of Technology, Nitte.

Karkala-574110

Program

```
#readlines( )
myfile=open("example.txt", "r")
line1 = myfile.readlines()
print("The first line is: ", line1[0])
print("The last line is: ", line1[-1])
```

Output

The first line is: Python Programming Elective.

The last line is: Karkala-574110

Program to count number of lines

Sample File



File Edit Format View Help

Python Programming Elective.
Department of Civil Engg.
NMAM Institute of Technology, Nitte.
Karkala-574110

Program

```
#Program to count number of lines
myfile=open("example.txt", "r")
a = myfile.readlines( )
lines = len(a)
print("The number of lines in the file = ", lines)
```

Output

The number of lines in the file = 4

Program to find size of the file in bytes

Sample File



File Edit Format View Help

Python Programming Elective.
Department of Civil Engg.
NMAM Institute of Technology, Nitte.
Karkala-574110

Program

```
#Program to find the size of the file in bytes
myfile=open("example.txt", "r")
a = myfile.read( )
size = len(a)
print("The size of the file in bytes is = ", size)
```

Output

The size of the file in bytes is = 106

```
>>> f = open("myfile.txt", 'r')
>>> for line in f:
        print(line)
```

First line.

Second line.

```
>>> f = open("myfile.txt", 'r')
>>> while True:
        line = f.readline()
        if line == "":
            break
        print(line)
```

First line.

Second line.

The **readline** method consumes a line of input and returns this string, including the newline. If **readline** encounters the end of the file, it returns the empty string.

Program to write total marks of students onto a file

```
#program to write marks of students using append "a" mode
myfile=open("marks.txt", "a")
ans='y'
while ans=='y':
    name=input("Enter student 1 name: ")
    sub1=int(input("Enter subject 1 marks: "))
    sub2=int(input("Enter subject 2 marks: "))
    sub3=int(input("Enter subject 3 marks: "))
    total = sub1 + sub2 + sub3
    a = "The total marks of " + name + " is " + str(total) + "\n"
    myfile.write(a)
    ans = input("add next student..?")
myfile.close()
```

Reading Numbers from a File

```
f = open("integers.txt", 'r')
theSum = 0
for line in f:
    line = line.strip()
    number = int(line)
    theSum += number
print("The sum is", theSum)
```

```
f = open("integers.txt", 'r')
theSum = 0
for line in f:
    wordlist = line.split()
    for word in wordlist:
        number = int(word)
        theSum += number
print("The sum is", theSum)
```

Writing number to a file

```
import random
f = open("integers.txt", 'w')
for count in range(500):
    number = random.randint(1, 500)
    f.write(str(number) + '\n')
f.close()
```

Working with CSV Files

Creating a CSV File

To create a CSV file, you can use Python's built-in `csv` module, which helps you easily write and read tabular data.

Reading a CSV File

The `csv.reader()` method reads data from a CSV file and returns it as rows (each row being a list of values).

Writing to a CSV File

The `csv.writer()` method writes data to a CSV file, with

- import csv
- # Open a CSV file in write mode
- file = open('data.csv', 'w', newline=')
- # Create a CSV writer object
- writer = csv.writer(file)
- # Writing a header (optional)
- writer.writerow(["Name", "Age", "City"])
- # Writing data rows
- writer.writerow(["Alice", 22, "New York"])

```
import csv

# Open the CSV file in read mode
file = open('data.csv', 'r')

# Create a CSV reader object
reader = csv.reader(file)

# Read and print each row
for row in reader:
    print(row)

# Close the file after reading
file.close()
```

```
import csv  
# Open the CSV file in append mode  
file = open('data.csv', 'a', newline=')
```

```
writer = csv.writer(file)
```

```
# Write a new row of data  
writer.writerow(["Charlie", 30, "Chicago"])
```

```
# Close the file  
file.close()
```

```
import csv
```

```
# First sheet: Enter 3 values
print("Enter 3 values for the first sheet:")
name = input("Name: ")
age = input("Age: ")
city = input("City: ")
```

```
# Write to the first CSV file (sheet1.csv)
file = open('sheet1.csv', 'a', newline="")
writer = csv.writer(file)
writer.writerow([name, age, city])
file.close()
```

```
# Second sheet: Enter 2 more values
print("\nEnter 2 values for the second sheet:")
country = input("Country: ")
job = input("Job: ")

# Write to the second CSV file (sheet2.csv)
file = open('sheet2.csv', 'a', newline="")
writer = csv.writer(file)
writer.writerow([country, job])

file.close()

print("\nData has been saved to 'sheet1.csv' and 'sheet2.csv'.")
```

Thank You

Unit 3

Introduction to Exceptions and Errors.

By,
Prithviraj Jain
CSE, NMAMIT

ERRORS

Error is an abnormal condition whenever it occurs execution of the program is stopped.

An error is a term used to describe any issue that arises unexpectedly that cause a computer to not function properly. Computers can encounter either software errors or hardware errors.

ERRORS

Errors are mainly classified into following types.

- 1. Syntax Errors**
- 2. Semantic Errors**
- 3. Run Time Errors.**
- 4. Logical Errors.**

1. Syntax Errors

Syntax errors occur when rules of a programming language are misused i.e., when a grammatical rule of the language is violated.

1. Syntax Errors

For instance in the following program segment,

```
def main()  
    a=10  
    b=3  
    print(" Sum is ",a+b
```

: (Colon) Missing

) Missing

2. Semantic Errors

Semantics error occur when statements are not meaningful

Semantics refers to the set of rules which give the meaning of the statement.

For example,

Rama plays Guitar

This statement is syntactically and semantically correct and it has some meaning.

2. Semantic Errors

**See the following statement,
Guitar plays Rama**

is syntactically correct (syntax is correct) but semantically incorrect. Similarly, there are semantics rules of programming language, violation of which results in semantical errors.

$$X * Y = Z$$

will result in semantical error as an expression can not come on the left side of an assignment statement.

3. Run Time Errors.

A Run time error is that occurs during execution of the program. It is caused because of some illegal operation taking place.

For example

1. If a program is trying to open a file which does not exists or it could not be opened(meaning file is corrupted), results into an execution error.
2. An expression is trying to divide a number by zero are RUN TIME ERRORS.

4. Logical Errors.

A Logical Error is that error which causes a program to produce incorrect or undesired output.

for instance,

```
ctr=1;
```

```
while(ctr<10):
```

```
    print(n *ctr)
```

Exceptions

Exceptions

What is an exception?

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called *exceptions*

Exceptions

For Example

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

Handling Exceptions

Handling Exceptions

It is possible to write programs that handle selected exceptions. These exceptions can be handled using the try statement:

try:

code that may cause exception

except:

code to run when exception occurs

Handling Exceptions

For Example

```
>>> while True:  
...     try:  
...         x = int(input("Please enter a number: "))  
...         break  
...     except ValueError:  
...         print("Oops! That was no valid number. Try again...")  
... 
```

Handling Exceptions

The try statement works as follows.

First, the try clause (the statement(s) between the try and except keywords) is executed.

If no exception occurs, the except clause is skipped and execution of the try statement is finished.

Handling Exceptions

The try statement works as follows.

If an exception occurs during execution of the try clause, the rest of the clause is skipped. Then if its type matches the exception named after the except keyword, the except clause is executed, and then execution continues after the try statement.

Handling Exceptions

If an exception occurs which does not match the exception named in the except clause, it is passed on to outer try statements; if no handler is found, it is an unhandled exception and execution stops with a message as shown above.

How does the try-except work

- First, the try block is executed.
- If any exception is not there in the code, then the try block will execute, except the block is finished.
- If an exception occurs then the except block will be executed and the try block is skipped.
- If an exception is still raised and the except block is left unhandled then it is passed to the outer try blocks. If still it is left unhandled then eventually the execution stops.
- A try statement can have more than one except block.

The *except* Clause with No Exceptions

The *except* Clause with No Exceptions

You can also use the except statement with no exceptions defined as follows:

try:

You do your operations here;

except:

If there is any exception, then execute this block.....

else:

If there is no exception then execute this block.

Contd..

Some common exceptions are:

- **SyntaxError:** This exception is raised when the interpreter encounters a syntax error in the code, such as a misspelled keyword, a missing colon, or an unbalanced parenthesis.
- **TypeError:** This exception is raised when an operation or function is applied to an object of the wrong type, such as adding a string to an integer.
- **NameError:** This exception is raised when a variable or function name is not found in the current scope.
- **IndexError:** This exception is raised when an index is out of range for a list, tuple, or other sequence types.

Some common exceptions are:

- **KeyError:** This exception is raised when a key is not found in a dictionary.
- **ValueError:** This exception is raised when a function or method is called with an invalid argument or input, such as trying to convert a string to an integer when the string does not represent a valid integer.
- **AttributeError:** This exception is raised when an attribute or method is not found on an object, such as trying to access a non-existent attribute of a class instance.
- **IOError:** This exception is raised when an I/O operation, such as reading or writing a file, fails due to an input/output error.
- **ZeroDivisionError:** This exception is raised when an attempt is made to divide a number by zero.
- **ImportError:** This exception is raised when an import statement fails to find or load a module.

IOError

If the file cannot be opened.

ImportError

If python cannot find the module.

ValueError

Raised when a built-in operation or function receives an argument that has the right type but an inappropriate value.

Some common exceptions

KeyboardInterrupt

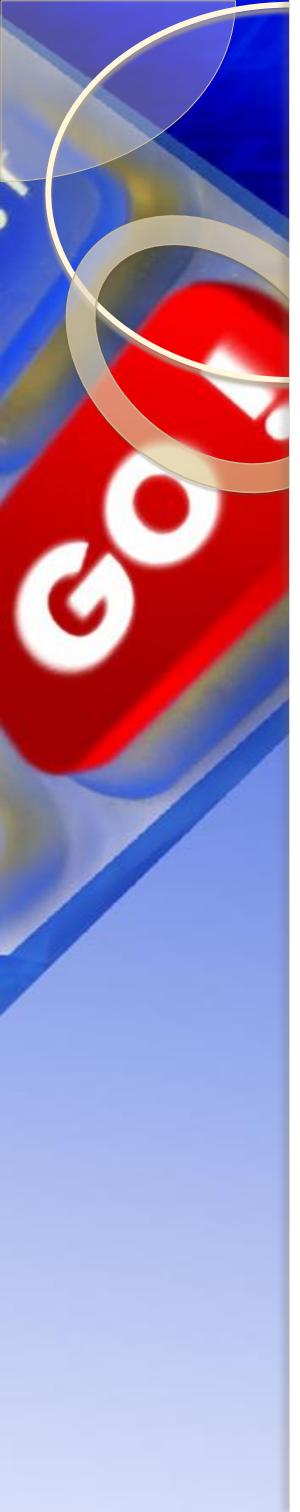
Raised when the user hits the interrupt key (normally Control-C or Delete).

EOFError

Raised when one of the built-in functions (`input()` or `raw_input()`) hits an end-of-file condition (EOF) without reading any data

Example

Example



```
Python 3.4.0: sam23.py - C:/Python34/sam23.py
File Edit Format Run Options Windows Help
def error_handling():
    try:
        print(1 / 0)
    except:
        print("Something bad happened")
error_handling()
Ln: 7 Col: 0
```

```
def divide(a, b):
    try:
        res = a // b
        print("Answer: ", res)
    except ZeroDivisionError:
        print("Error dividing it by zero")

divide(3, 2)
```

Output

Answer: 1



raise statement

raise statement

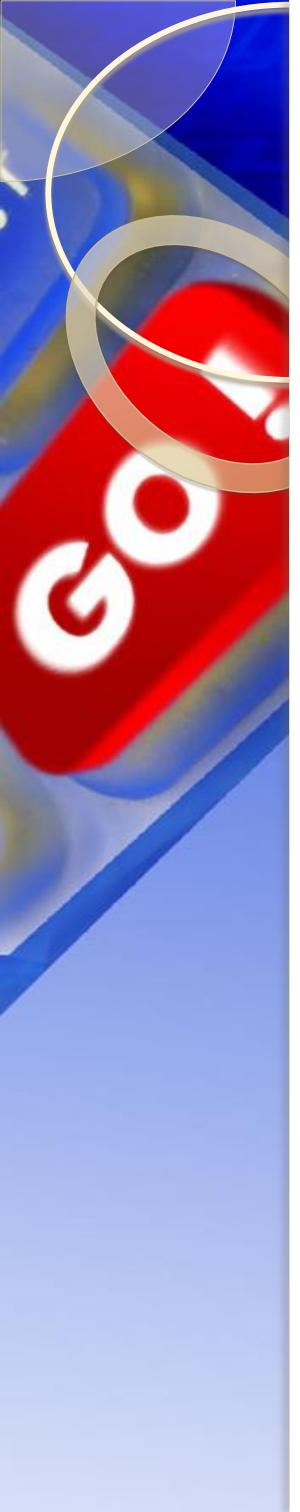
You can raise an exception in your own program by using the raise exception.

Raising an exception breaks current code execution and returns the exception back until it is handled.

Syntax:

raise [expression1[, expression2]]

raise Example



```
Python 3.4.0: a1.py - C:/Python34/a1.py
File Edit Format Run Options Windows Help
a = "hi"
if not type(a) is int:
    raise TypeError("Only integers are allowed")
Ln: 2 Col: 0
```

raise Example



```
Python 3.4.0: a1.py - C:/Python34/a1.py
File Edit Format Run Options Windows Help
a = "hi"
if not type(a) is int:
    raise TypeError("Only integers are allowed")
Ln: 2 Col: 0
```

Else and Finally

1

try:

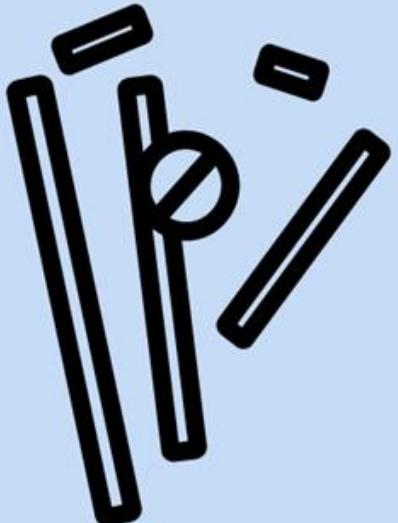
try to hit the ball

3

else:

else it will go in the hands of wicket keeper.

2

except:If didn't hit,
then you will get
bowled.

4

finally:

Irrespective of prior cases, the ball will be thrown.

Introduction

- In python, an error can be of two types:
 - Syntax errors
 - Exceptions

Syntax errors: Errors are defined as the problems that cause the program to stop its execution.

Exceptions: Exceptions cause a change in the normal flow of the program due to some internal events that occurred in the program.

Difference between Syntax Error and Exceptions

Syntax Error: This error is caused by the wrong syntax in the code.

```
if(amount > 999)
    print("amount more than 1000")
```

```
if(amount > 999)
```

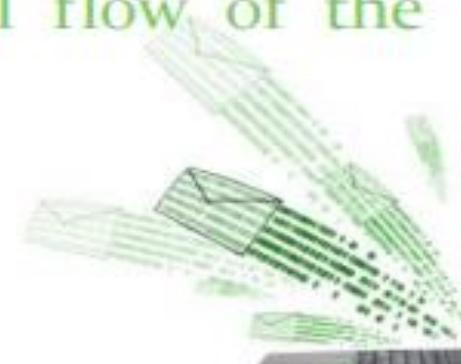
```
^
SyntaxError: invalid syntax
```

Exceptions: Exceptions are raised when the program is syntactically correct, but the code results in an error. This error does not stop the execution of the program, however, it changes the normal flow of the program.

```
a = 10 / 0
```

```
print(a)
```

```
ZeroDivisionError: division by zero
```



What does it do?

- ✖ Tries to run a block of code.
- ✖ If an error occurs, it catches and handles it.

- Example:

```
try:  
    x = 10 / 0  
except ZeroDivisionError:  
    print("You can't divide by zero!")
```

- Output: You can't divide by zero!

An exception is present, so the except block will run.

```
def divide(a, b):
    try:
        res = a // b
        print("Answer: ", res)
    except ZeroDivisionError:
        print("Error dividing it by zero")
```

```
divide(3, 0)
```

Output

```
Error dividing it by zero
```

Handling Multiple Errors

What if different errors can occur?

Use multiple `except` blocks.

- Example:

```
try:  
    num = int("hello") # This will cause a ValueError  
except ValueError:  
    print("That's not a number!")  
except TypeError:  
    print("Wrong type of data!")
```

- Output: That's not a number!

Else Block

- An else block can also be added on the try-except block and it should be present after all the except blocks. If the try block successfully does not raise an exception then the code enters into the else block.

Syntax

```
try:  
    # Some Code to check for errors  
except:  
    # if an error in the  
    # try block then this block runs  
else:  
    # executes this block if  
    # there is no exception in the try block
```

The else Clause

What is it for?

- Runs only if no error happens in `try` block.
 - Example:

```
try:  
    num = int("100")  
except ValueError:  
    print("Invalid input!")  
else:  
    print("valid number:", num)
```

- Output: Valid number: 100

Code

```
def calc(x , y):
    try:
        z = ((x+y) // (x-y))
    except ZeroDivisionError:
        print ("The divide is resulting in 0")
    else:
        print (z)

calc(2.0, 3.0)
calc(3.0, 3.0)
```

Output

-5.0

The divide is resulting in 0

Finally Keyword

- In Python, we use the final keyword, which is executed always even if the exception is not handled and the try-except block is terminated. It is executed after the try-except block.

Syntax

```
try:  
    # Some Code to check for errors  
except:  
    # If an error in the  
    # try block then execute this block  
else:  
    # If no exception then execute this  
finally:  
    # This code always executed after the try-except
```

The finally Clause

What is it for?

- Runs no matter what, useful for cleanup tasks.

- Example:

```
try:  
    f = open("test.txt", "r")  
except FileNotFoundError:  
    print("File not found!")  
finally:  
    print("Execution completed.")
```

- Output:

- If file is missing: File not found! then Execution completed.
- If file is present: Execution completed.

try:

Code to execute

except:

Code to execute in case of error

else:

Code to execute in case NO error

finally :

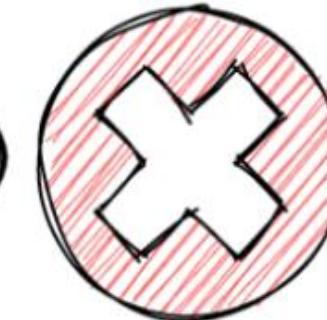
Code to execute in all cases



try

except

else



finally



Code

```
try:  
    k = 5//0  
    print(k)  
  
except ZeroDivisionError:  
    print("ZeroDivisionError")  
  
finally:  
    print("The code under finally is always executed")
```

Output

```
ZeroDivisionError  
The code under finally is always executed
```



Thank You



Introduction to Pandas Data Visualization using Matplotlib

Unit 3

What is Pandas?

- × **Pandas is a Python library** used for data manipulation and analysis.
- × **Provides two main data structures** – Series (1D) and DataFrame (2D).
- × **Designed for structured data** like CSV, Excel, and SQL databases.
- × **Supports data cleaning** – handling missing values, filtering, and aggregation.
- × **Allows easy data transformation** like sorting, merging, and grouping.
- × **Works well with NumPy and Matplotlib** for efficient data analysis and visualization.

What is Matplotlib?

- × **Matplotlib is a Python library** used for creating visualizations like graphs and charts.
- × **Supports static, animated, and interactive plots** for data analysis.
- × **Commonly used plots** include line plots, bar charts, scatter plots, histograms, and pie charts.
- × **Highly customizable** – allows setting labels, titles, colors, and legends.
- × **Works well with Pandas and NumPy** for easy data visualization.
- × **Can save plots as images** in formats like PNG, JPG, and PDF.

Using Pandas with Matplotlib

```
import pandas as pd  
import matplotlib.pyplot as plt  
  
# Load data  
data = pd.DataFrame({'Year': [2020, 2021, 2022], 'Sales': [200, 250, 300]})  
  
# Visualize  
data.plot(x='Year', y='Sales', kind='line', title='Yearly Sales')  
plt.show()
```

Creating a Pandas Series

```
python
```

```
import pandas as pd  
  
data = pd.Series([10, 20, 30, 40])  
  
print(data)
```

- Output:

```
go  
  
0    10  
1    20  
2    30  
3    40  
dtype: int64
```

Creating a Pandas Series

1. **Creates a dictionary** with two keys: '**Name**' and '**Age**', storing lists of values.
2. **Converts the dictionary into a DataFrame** using `pd.DataFrame()`, organizing data in a tabular format.
3. **Prints the DataFrame**, displaying names and ages in a structured table with automatic indexing.

Creating a Pandas DataFrame

```
python
```

```
import pandas as pd  
  
data = {'Name': ['Alice', 'Bob', 'Charlie'], 'Age': [25, 30, 35]}  
  
df = pd.DataFrame(data)  
  
print(df)
```

- Output:

```
markdown
```

	Name	Age
0	Alice	25
1	Bob	30
2	Charlie	35

Creating a Pandas DataFrame

1. **Creates a Pandas Series** – A one-dimensional labeled array storing values [10, 20, 30, 40].
2. **Auto-indexing** – Each value gets an index starting from 0 (default).
3. **Prints the Series** – Displays values along with their index in a structured format.

Reading Data from a CSV File

```
python  
  
df = pd.read_csv('data.csv')  
print(df.head()) # Display first 5 rows
```

1. **Reads a CSV file** named 'data.csv' and loads it into a Pandas DataFrame.
2. **df.head()** **displays the first 5 rows** of the DataFrame for a quick preview of the data.
3. **Helps in understanding the dataset structure**, including columns and initial values.

Data Cleaning and Preprocessing

- Handling missing values:

python

```
df.fillna(0) # Replace NaN with 0  
df.dropna() # Remove rows with NaN values
```

- Filtering data:

python

```
df[df['Age'] > 30]
```

- Adding new columns:

python

```
df['Salary'] = [50000, 60000, 70000]
```

Data Cleaning and Preprocessing

1. **Handling Missing Data** – `fillna(0)` replaces missing values with 0, and `dropna()` removes rows with missing values.
2. **Filtering Data** – `df[df['Age'] > 30]` selects rows where the `Age` column has values greater than 30.
3. **Adding a New Column** – `df['Salary'] = [50000, 60000, 70000]` inserts a new column named "Salary" into the DataFrame.

Using Pandas with Matplotlib

Pandas integrates with Matplotlib for seamless visualization.

Workflow:

- Load data using Pandas.
- Perform analysis and transformations.
- Visualize using Matplotlib.

Basic Line Plot in Matplotlib

```
python
```

```
import matplotlib.pyplot as plt  
  
x = [1, 2, 3, 4, 5]  
y = [10, 20, 25, 30, 40]  
plt.plot(x, y)  
plt.xlabel('X-axis')  
plt.ylabel('Y-axis')  
plt.title('Line Plot Example')  
plt.show()
```

Basic Line Plot in Matplotlib

1. **Creates a line plot** using `plt.plot(x, y)`, where `x` represents the X-axis values and `y` represents the Y-axis values.
2. **Adds labels and a title** using `xlabel()`, `ylabel()`, and `title()` to describe the plot.
3. **Displays the plot** using `plt.show()`.

Bar Chart using Matplotlib

```
python
```

```
categories = ['A', 'B', 'C', 'D']
values = [10, 20, 15, 25]
plt.bar(categories, values, color='blue')
plt.title('Bar Chart Example')
plt.show()
```

Bar Chart using Matplotlib

1. **Creates a bar chart** using `plt.bar(categories, values)`, where categories `['A', 'B', 'C', 'D']` are on the X-axis and their values `[10, 20, 15, 25]` are on the Y-axis.
2. **Sets the bar color to blue** and adds a title using `plt.title()`.
3. **Displays the bar chart** using `plt.show()`.

Key Benefits

- **Pandas simplifies data manipulation** by providing efficient tools for handling structured data.
- **Matplotlib helps visualize data** through charts and graphs, making patterns easier to understand.
- **Together, they enable exploratory data analysis** by allowing users to clean, analyze, and visualize data seamlessly.
- **Supports various data formats** like CSV, Excel, and SQL for easy integration.
- **Helps in data-driven decision-making** by presenting insights clearly through graphs.
- **Widely used in data science and analytics** for tasks like trend analysis and pattern recognition.

Thank You!