Define Python

Sanjana. Menashi

INTRODUCTION TO PYTHON

Def 2M

- **Python :** Python is a general-purpose interpreted, interactive, object-oriented, and high-level programming language that can be used effectively to build almost different types of applications. It was created by **Guido van Rossum, and released in 1990.**

- **Features of Python :** / Advantages.

5M

If they asked only mention the features or Explain them

  - **Python is Interpreted** – Python is processed at runtime by the interpreter. We do not need to compile the program before executing it. This is similar to PERL and PHP.
  - **Python is Object-Oriented** – Python supports Object-Oriented style or technique of programming.
  - **Python is a Beginner's Language** – Python is a great language for the beginner-level programmers and supports the development of a wide range of applications from simple text processing to browsers.
  - **Easy-to-learn** – Python has few keywords, simple structure, and a clearly defined syntax. This allows the user to pick up the language quickly.
  - **Easy-to-read** – Python code is more clearly defined and visible to the eyes.
  - **Easy-to-maintain** – Python's source code is fairly easy-to-maintain.
  - **A broad standard library** – Python's bulk of the library is very portable and cross-platform compatible on UNIX, Windows, and Macintosh.
  - Cross-platform Language/ Portable - Python **works on different platforms** (Windows, Mac, Linux, Raspberry Pi, etc).
  - **Databases** – Python provides interfaces to all major commercial databases.
  - **GUI Programming Support** - Python supports GUI applications that can be created and ported to many system calls, libraries and windows systems, such as Windows MFC, Macintosh, and the X Window system of Unix.

- **History of Python**
  - Python was developed by Guido van Rossum in 1990 at the National Research Institute for Mathematics and Computer Science in the Netherlands.It is derived from many other languages, including ABC, Modula-3, C, C++, Algol-68, SmallTalk, Unix shell and other scripting languages. It gained popularity with the arrival of Python 2.0 in 2000. In incorporated a number of important improvements to the language and added libraries. Python 3.0 was released at the end of 2008. Python is copyrighted. Python source code is now available under the GNU General Public License (GPL). Latest version is 3.10

Appl^n of python.

- GUI based desktop appl^n.
- Graphic design
- Operating system
- Web frameworks & appl^n
- Database Access
- Education
- Software development
- Language development

The Basic Elements of Python

- **Python Program -** It is also called a **script**, is a sequence of definitions and commands.
- **Shell –** Shell is a command interpreter. The definitions are evaluated and the commands are executed by the **shell**. A new shell is created whenever execution of a program begins.
- **Command** – It is also called a **statement**. It instructs the interpreter to do different tasks.

Comments in Python

❖ **Python Operators –** Python supports a number of operator types, namely:

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

- **Arithmetic Operators-** they are used to perform arithmetic operations

Symbol	Meaning	Example	Result if i=5 and j=2
+	Addition	i+j	7
-	Subtraction	i–j	2
*	Multiplication	i*j	10
//	Integer	i//j	2
/	Division	i/j	2.5
%	Modulus	i%j	1
**	Exponent	i**j	25

- **Relational Operators-** they are used to perform comparison operations

Operator	Meaning	expression	Result if a=10 and b=20
==	Equal to	a==b	false
!= <>	Not Equal to	a!=b	true
>	Greater than	a>b	false
<	Less Than	a<b	true
>=	Greater Than or equal to	a>=b	false
<=	Less than or equal to	a<=b	true

- **Logical Operators-** they are used to perform comparison operations

Operator	Meaning	expression	Result if a=10, b=20, p=30,q=40
and	Logical And – return true when both the expressions are true	(a<b)and(p<q)	true
or	Logical OR – return true when any one of the expressions is true	(a>b)and(p<q)	True
not	Logical NOT- negates the expression	not(a<b)	False

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

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

- **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	Meaning	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

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

Operator	Meaning	example
in	Returns True if a sequence with the specified value is present in the object	x in y
not in	returns True if a sequence with the specified value is not present in the object	x not in y

```
a=b
b=temp
print("After swapping a = ",a,"b= ",b)
```

❖ **Programming Exercises:**

1. Write a Python Program to find the surface area of a cylinder
2. Write a Python Program to convert temperature from celcius to farenheit
3. Write a Python Program to find the area of a cone
4. Write a Python Program to calculate displacement of an object
5. Write a Python Program to compute Simple Interest

❖ **Control Structures:** Control Structures are used to control the flow execution of a program. Python provides following control structures(way of computation):

- Straight line programs (Sequential)
- Branching programs :

❖ **Conditional Execution -** Branching statements are also called conditional statements. A conditional statement has three parts

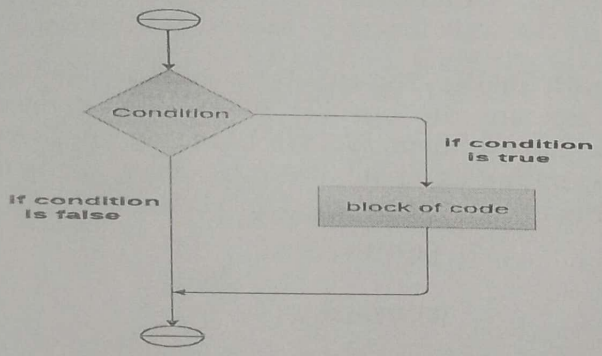
1. A test
2. Block of code to be executed if test is true
3. Optional block of code if test is false

There four types of if statements, viz,

❖ **The if statement -** The if statement is used to test a particular condition and if the condition is true, it executes a block of code known as if-block. The condition of if statement can be any valid logical expression which can be either evaluated to true or false.

❖ **Syntax:**

```
if expression:
        statement
```



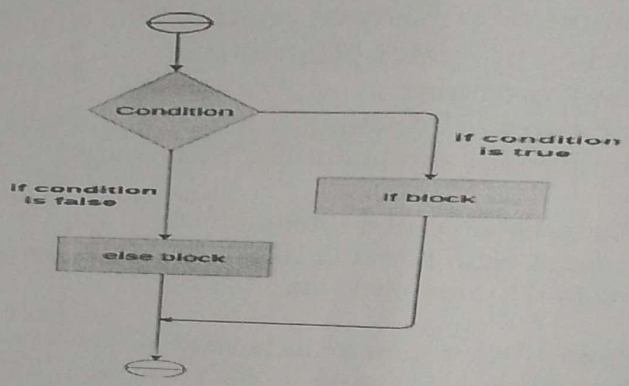
❖ Example 1

```
num = int(input("enter the number "))
if num%2 == 0:
    print("Number is even")
```

- **The if-else statement** -The if-else statement provides an else block combined with the if. In this form if the condition is true the if block is executed otherwise the else part is executed .

- **Syntax:**

```
if condition:
        #block of statements
else:
        # block of statements
```

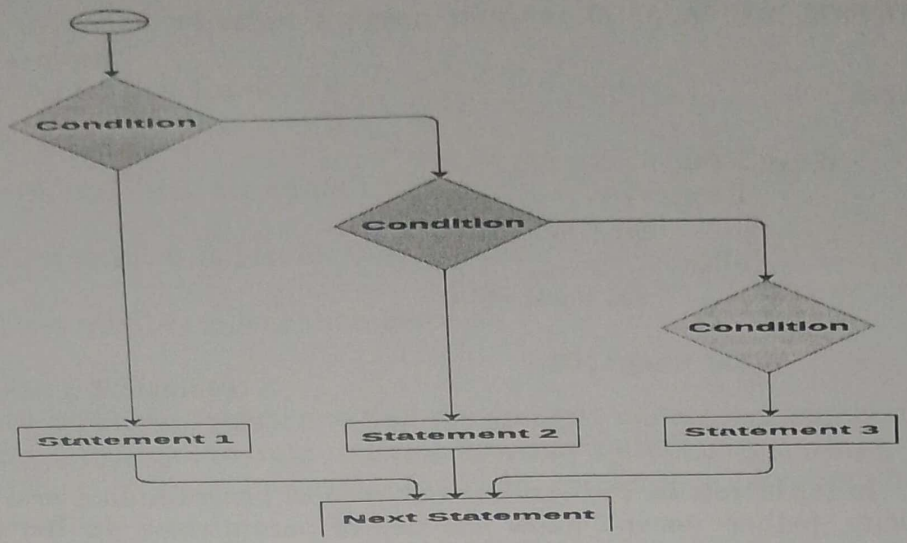


- Example 1 : Program to check whether a person is eligible to vote or not.

```
age = int (input("Enter your age? "))
if age>=18:
   print("You are eligible to vote ")
else:
   print("Not eligible to vote")
```

- **The elif statement (Chained Conditionals)** -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. However, using elif is optional. The elif statement works like an if-else-if ladder statement in C.
- **Syntax :**

```
if expression 1:
   # block of statements
elif expression 2:
   # block of statements
elif expression 3:
   # block of statements
else:
   # block of statements
```



❖ Example 1

```
marks = int(input("Enter the marks? "))
if marks > 85:
        print("Congrats ! you scored grade A ...")
elif marks > 60:
        print("You scored grade B + ...")
elif marks > 40:
        print("You scored grade B ...")
elif marks > 30:
        print("You scored grade C ...")
else:
        print("Fail")
```

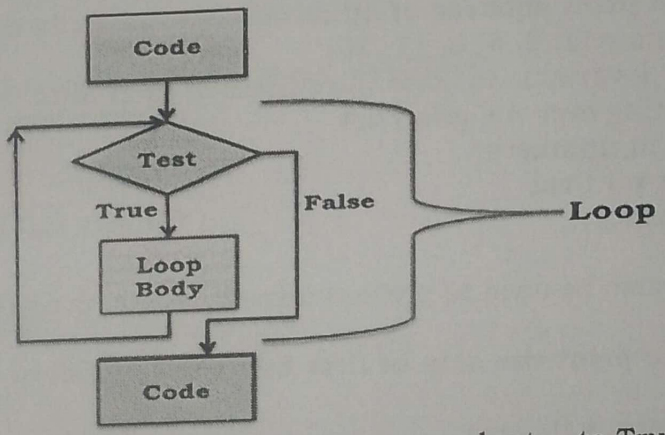
The nested if statement (Nested Conditionals) - You can have if statements inside if statements, this is called *nested* if statements. There may be a situation when you want to check for another condition after a condition resolves to true. In such a situation, you can use the nested if construct.

Syntax :

```
if expression1:
        if expression2:
                statement(s)
        else:
                statement(s)
else:
                statement(s)
```

In a nested if construct, you can have an if...elif...else construct inside another if...elif...else construct

- **Iteration/Looping:** It is a process of executing set of instructions repeatedly until a given condition is satisfied.



Looping statement begins with a test. If the test evaluates to True, the program executes the loop body once, and then goes back to reevaluate the test. This process is repeated until the test evaluates to False, after which control passes to the code following the iteration statement. The different looping statements in Python are: For loop and while loop

- **for loop** - for loop executes the block of statements until the given condition is satisfied. It is a entry controlled or a pre-tested loop. It is better to use for loop if the number of iteration is known in advance.

Syntax:

```
for <variable> in <sequence>:
        body_of_loop
```

Here <variable> is a variable that is used for iterating over a <sequence>. On every iteration it takes the next value from <sequence> until the end of sequence is reached.

- **Example:**

```
str = "Hello"

for i in str:
    print(i)
```

Output:

```
H
e
l
l
o
```

for loops are commonly used to process lists.

 # Program to print squares of all numbers present in a list

```
numbers = [1, 2, 4, 6, 11, 20]
sq = 0  # variable to store the square of each num temporary
# iterating over the given list
for val in numbers:
  sq = val * val
  print(sq)
```

for loop can also be used to process a range of values using range function

 # Program to print the sum of first 5 natural numbers

```
sum = 0
for val in range(1, 6):
    sum = sum + val
print(sum)
```

- **While loop:** It is also a pre-tested loop. In the while loop, the block of statements is executed until the specified condition is satisfied.

Syntax:

```
while test_expression:
  Body of loop
```

In the above syntax, test expression is checked first. The body of the loop is entered only if the test_expression evaluates to True. After one iteration, the test expression is checked again. This process continues until the test_expression evaluates to False. The while loop is to be used in the scenario where we don't know the number of iterations in advance. In Python, the body of the while loop is determined through indentation. Body starts with indentation and the first unindented line marks the end of the block.

```
# Program to add natural numbers upto sum = 1+2+3+...+n
n = int(input("Enter n: "))
sum = 0
i = 1
while i <= n:
   sum = sum + i
   i = i+1
print("The sum is", sum)
```

- **Programs based on looping**

```
#Program to display even numbers between 1 to n
n=int(input("Enter the number :"))
for i in range(1,n+1):
   if (i%2)==0:
      print(i)
```

```
#Program to find the factorial of a given number
n = int(input('Enter a number: '))

if n<0:
   print('Enter a positive number')
else:
   fact = 1
   for i in range(1, n+1):
      fact = fact * i
   print('Factorial of', n, '=', fact)
```

```
# Program to print first n fiboncii series
n=int(input("Enter value of n"))
print("Fibonacii series is:")

f1=0
f2=1
print(f1)
print(f2)

for i in range(3,n+1):
```

①Defn functions & its Types ②Explain the types. ③Give an example for function.

FUNCTIONS

- **Python Functions**

Python allows us to divide a large program into the basic building blocks known as function. A function is subprogram or a subroutine that performs a particular task. A function can be defined as the organized block of reusable code which can be called whenever required.A function can be called multiple times to provide reusability and modularity to the python program.
Functions can be divided into 2 types.

1. **Built-in Functions:** These are in built functions defined in Python. They can be called in any program. Eg. range(), print(), len etc
2. User Defined Functions: The functions defined by the user or programmer are called user-defined functions.

- **Decalaring (Creating) a function**

In python, we can use **def** keyword to define the function.

- **Syntax**

```
def function_name(list of formal parameters):
    """ doc string""""
    Function_suite
    return <expression>
```

The function block starts with the colon (:). Generally we should write a string as the first statement in the function body. This sstatement is known as the doc string that gives information about the function. However, the doc string is optional. The function suite consists of statements in that make up the function body. These block of statements is indented including the return statement. The last statement in the function block is the **optional** return statement that specifies the value returned by the function.

Example:

function Example.

```
#Function definition
def sum( a,b ):
    """ This function find sum""""
    c=a+b
    return(c)
```

Calling a function

In python, a function must be defined before the function calling otherwise the python interpreter gives an error. Once the function is defined, we can call it from another function or the python prompt. To call the function, use the function name followed by the parentheses.

grocery("sugar")

❖ **Void and fruitful functions** 2M Differentiate

Void Functions	Fruitful Functions
Do not return a value	Always return a value
No need to assign it to a variable	Must assign it to a variable to hold return value
return statement may or may not be present	return statement is always present
Example def greeting(name) print ("hello") return	Example def add(a, b) c=a+b return c

2M ❖ **Recursion**

Recursion is a process in which a function calls itself repeatedly till certain condition is met. Functions that incorporate recursion are called recursive functions.

❖ **Examples:**

#Recursive factorial function

```
def fact(n):
    if n==0 or n==1:
        return 1
    else:
        return n*fact(n-1)

n=int(input("Enter the value of n"))

f=fact(n)
print("Factorial of",n,"=",f)
```

#Recursive function to compute the value of a^n

```
def power(a,n):
    if n==0:
        return 1
    elif n==1:
        return a
    else:
        return a*power(a,n-1)

a=int(input("Enter the vlue of base"))
n=int(input("Enter the value of exponent"))

f=power(a,n)
```

```
print(a,"to the power of",n,"=",f)
```

#Recursive function to generate fibonacii numbers

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)

n=int(input('How many elements?'))
for i in range(0,n):
    print(fib(i))
```

Advantages of Recursion:

1. A recursive code has a cleaner-looking code.
2. Recursion makes it easier to code, as it breaks a task into smaller ones.
3. It is easier to generate a sequence using recursion than by using nested iteration

❖ **Scope**

Defn:Scope defines portion of programs where we can access a particular identifier. The scopes of the variables depend upon the location where the variable is being declared.

In python, the variables are defined with the two types of scopes.

1. **Global variables** : The variable defined outside any function is known to have a global scope Global variables can be accessed throughout the program

```
def strfun():
    print (s)

# Global scope
s = "I like Python"
strfun()
print (s)
```

Output:
I like Python
I like Python

In the above program s is global string variable that can be accessed anywhere, Hence it produces the above output.

2. **Local variables** : The variable defined inside a function is known to have a local scope. Such variables cannot be accessed outside the function.

IDE
→ Spider
→ Jupyter NoteBook.
→ Pycharm
→ visual studio code.

```
def strfun():
    s= "i like python"
    print(s)

#main program
strfun()
print(s)
```

Output:
i like python
NameError: name 's' is not defined

In the above program s is local string variable that is defined inside the function. It can be accessed only inside the function. Hence it produces an error.

Modules

Usually we store entire program code in one file. When programs get larger and multiple people work on it, it is more convenient to store different parts of the program in different files. Python modules allow us to easily construct a program from code in multiple files.

- **Defn :** A module is a file containing Python function definitions and statements. It is saved with **.py** extention. Modules in Python provides us the flexibility to organize the code in a logical way. To use the functionality of one module into another, we must have to **import** the specific module.

- **Loading the module in our python code**
We need to load the module in our python code (program) to use its functionality. Python provides two types of import statements as defined below.
1. *The import statement*
2. *The from-import statement*
- **The import statement :**
The import statement is used to import all the functionalities (functions) of one module into another. We can import multiple modules with a single import statement.
Syntax:
import module1,module2,..... module n

The from-import statement

Explain the user-defined function with an example.

→ A user-defined function is a function that is created by the user to perform a specific task.

Syntax

```
def function_name(parameters):
    statements(s)
    return value.
```

Example-

```
def square(num):
    result = num*num
    return result.
print(square(5)) → output → 25
print(square(8)).  "  → 64
```

② What are command line arguements? Give an example.

→ These are inputs passed to a python script when it is executed from the command line.

Ex:-

```
If len(sys.argv) < 2:
    print("Usage: python example.py <name")
else:
    name = sys.argv[1]
    print(f"Hello, {name}!")
```

Sanjana Menashi 244

Python Tuple

Creating a Dictionary

- A **dictionary** is a data structure that holds **key:value** pairs.
- The items should be enclosed in curly braces – { }.
- **Key** is immutable and is unique, whereas, **value** is mutable and can be duplicated.
- Dictionary is unordered. **Values** are accessed by using **keys**.
- Dictionary can contain heterogeneous items.

Examples:

```
student = {'name':'Sachin', 'rno':201, 'per':75.5}
digit_word = {1:'one', 2:'two', 3:'three', 4:'four'}
my_car = {'make':'Tata', 'model':'Nexon', 'year': 2020,
'electric':True}
```

Operations on Dictionary

Accessing

The value of an item in a Dictionary can be accessed by using the key inside a pair of **square brackets [ ]**.

```
1 d = {1:'one', 2:'two', 3:'three'}
2 print(d[2])
```

```
two
```

Modifying

The value of an item in a Dictionary can be modified as below:

```
1 d = {1:'one', 2:'two', 3:'three'}
2 print(d)
3 d[3] = 'THREE'
4 print(d)
```

```
{1: 'one', 2: 'two', 3: 'three'}
{1: 'one', 2: 'two', 3: 'THREE'}
```

Traversing

Items of a dictionary can be traversed using for loop. We can access value associated with each key as given in below example.

Built-in Function on Dictionary

The basic built-in functions of python such as `len()`, `max()`, `min()`, `sum()`, `sorted()` can also be applied to dictionaries. Consider the following examples:

```
d = {4:'four', 2:'two', 1:'one', 3:'three'}
```

Function	Meaning	Example	Return value
len()	It accepts dictionary object as argument and returns the total number of items present in the dictionary.	`len(d)`	4
max()	It accepts dictionary object as argument and return the largest key. *Data type of all the keys must be same.*	`max(d)`	4
min()	It accepts dictionary object as argument and return the smallest key. *Data type of all the keys must be same.*	`min(d)`	1
sum()	It accepts dictionary object as argument and returns the sum of all keys in the dictionary. *Data type of keys must be int/float.*	`sum(d)`	10
sorted()	It accepts dictionary object as argument and returns a **new list** containing all keys arranged in ascending order. *Data type of keys must be same. The original dictionary is not modified.*	`sorted(t)`	[1, 2, 3, 4]

Dictionaries are mutable, but [illegible]
class provides several built-in methods.

```
d = {1:'one', 2:'two', 3:'three', 4:'four'}
```

Function	Meaning	Example	Dictionary contents / Return value
items()	Returns the dictionary items in **key:value** pair	`d.items()`	dict_items([(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')])
keys()	Returns key names in a dictionary.	`d.keys()`	dict_keys([1, 2, 3, 4])
values()	Returns value names in a dictionary.	`d.values()`	dict_values(['one', 'two', 'three', 'four'])
get()	Returns the value of a specified key in a dictionary. If key, does not exist, nothing will be retrieved.	`d.get(2)`	'two'
update()	Updates a value of a specified key in a dictionary. If specified key does not exist, then it adds as a new item into the dictionary.	`d.update({3:'THREE', 5:'five'})`	{1: 'one', 2: 'two', 3: 'THREE', 4: 'four', 5: 'five'}
pop()	Removes the item with specified key name	`d.pop(2)`	'two' {1: 'one', 3: 'THREE', 4: 'four', 5: 'five'}
popitem()	Removes any random (usually last inserted item) from the dictionary.	`d.popitem()`	(5, 'five') {1: 'one', 3: 'THREE', 4: 'four'}
copy()	Create a copy of dictionary	`d.copy()`	{1: 'one', 3: 'THREE', 4: 'four'}
clear()	Removes all items from a dictionary.	`d.clear()`	{}

Regular Expression

Definition of Regular Expression

Regular Expression or RegEx is sequence of characters that are used to search for a pattern in a string. In Python, **re** module provides support to regular expressions. Regular expression can be formed by combining:

- meta-characters
- special sequences
- sets

Functions of Regular Expression 5M or 10M

The **re** modules provides various functions to match/search a specific pattern in a string. The following are some of the functions:

- `match()`
- `search()`
- `findall()`
- `sub()`
- `split()`

match() Function

This function is used to search a pattern at the beginning of the string. If match is found, then it returns a match object.

Syntax:

```
re.match(pattern, string)
```

where, **pattern** : Regular expression to be matched.

string : String where pattern is searched

If match is True, then it returns a match object that contains mixed information about the search result. To extract the required part from the result, the following methods are used:

- **group()** - The part of the string is returned where the match is found
- **start()** - displays the starting position (index) of the searched pattern
- **end()** - displays the ending position (index) of the searched pattern. The ending index is excluded.
- **span()** - It returns the tuple containing the starting and end position (excluded) of the match.
- **string** - It returns a string passed into the match() function.

Example:

```
import re
s = 'madam, I am good madam'
res = re.match('madam', s)
```

```
print(res)
print('Group : ', res.group())
print('Start : ', res.start())
print('End : ', res.end())
print('Span : ', res.span())
print('string : ', res.string)
```

Output:

```
<re.Match object; span=(0, 5), match='madam'>
Group :  madam
Start :  0
End :  5
Span :  (0, 5)
string :  madam, I am good madam
```

search() Function

This function is used to search a pattern at any position in a string. It stops searching after the first match i.e. it only returns the first occurrence of the search pattern.

Syntax:

```
re.search(pattern, string)
```

where, **pattern** : Regular expression to be matched.

string : String where pattern is searched

If search is True, then it returns a match object that contains mixed information about the search result. To extract the required part from the result, the following methods are used:

- **group()** - The part of the string is returned where the match is found
- **start()** - displays the starting position (index) of the searched pattern
- **end()** - displays the ending position (index) of the searched pattern. The ending index is excluded. The ending index is excluded.
- **span()** - It returns the tuple containing the starting and end position (excluded) of the search.
- **string** - It returns a string passed into the match() function.

Example:

```
import re
s = 'madam, I am good madam'
res = re.search('am', s)
print(res)
print('Group : ', res.group())
print('Start : ', res.start())
print('End : ', res.end())
print('Span : ', res.span())
print('string : ', res.string)
```

Output:

```
<re.Match object; span=(3, 5), match='am'>
Group :  am
Start :  3
End :  5
Span :  (3, 5)
string :  madam, I am good madam
```

findall() Function

This function returns a list containing all occurrence of a pattern from a string. If there are no matches, then an empty list is returned.

Syntax:

```
re.findall(pattern, string)
```

where, **pattern** : Regular expression to be matched.

string : String where pattern is searched

Example:

```
import re
s = 'madam, I am good madam'
res = re.findall('am', s)
print(res)
```

Output:

```
['am', 'am', 'am']
```

split() Function

This function splits a string by the occurrences of given pattern. It returns a list in which the string has been split in each match.

Syntax:

```
re.split(pattern, string)
```

where, **pattern** : Regular expression to be matched.

string : String where pattern is searched

Example:

```
import re
s = 'madam, I am good madam'
res = re.split('am', s)
print(res)
```

Output:

```
['mad', ', I ', ' good mad', '']
```

Exception Handling

Definition: Exceptions are the errors that occur during run-time of a program. When exception occurs, Python forces program to output an error and terminates the program execution abruptly.

Python has two types of exceptions, namely, built-exceptions and user defined exceptions.

Built-in Exceptions

The built-in exceptions are predefined in Python library. The built-in exceptions are raised when specific run-time error occurs. Programmer can also raise these exceptions using **`raise`** and **`assert`** keywords.

Python has several built-in exceptions and some of those are listed below:

Exception	Meaning
IndexError	Raised when an index of a sequence does not exist
NameError	Raised when a variable does not exist
TypeError	Raised when two different types are combined
ValueError	Raised when there is a wrong value in a specified data type
ZeroDivisionError	Raised when the second operator in a division is zero
KeyError	Raised when key does not exist in the dictionary
KeyboardInterrupt	Raised when the user presses Ctrl+c, Ctrl+z or Delete
AssertionError	Raised when an assert condition fails
EOFError	Raised when user presses Ctrl+d
Exception	Base class for all exceptions

Exception handling can be implemented in Python by using **`try`**, **`except`**, **`else`**, **`finally`** blocks. The blocks **`try`** and **`except`** are mandatory, whereas **`else`** and **`finally`** are optional.

`try` Block

The **`try`** block allows programmer to test a block of code for exceptions. The exception is raised (thrown) and program stops execution, if error occurs, otherwise all the statements are executed. The **`try`** block should follow at least one **`except`** block.

`except` Block

The **`except`** block allows programmer to handle the exceptions, if they are raised by the try block, otherwise this block will be skipped from execution. Programmer can write multiple **`except`** blocks for various types of exceptions. If the exception type is unknown, then the base class **`Exception`** can be used.

else Block

The **else** block allows programmer to execute a set of statements if no exceptions are raised, otherwise, is skipped from execution.

finally Block

The **finally** block allows programmer to execute a set of statements regardless whether exceptions are raised or not.

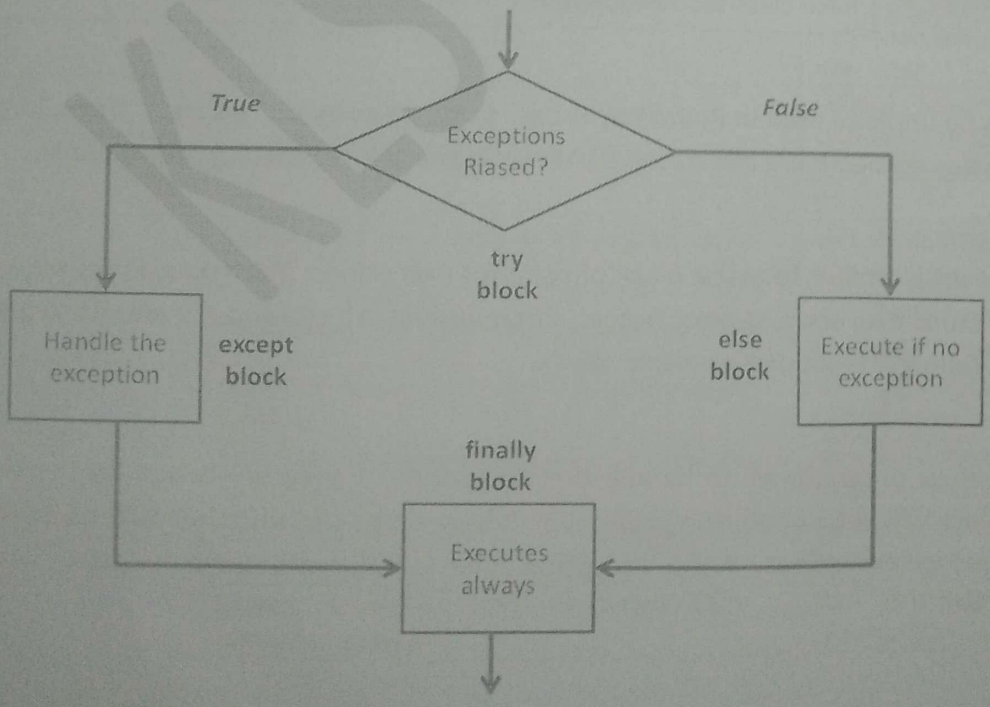
Explain them and write a journal program.

Syntax: Exception Handling in Python

```
try:
    Set of statements
    that may raise exceptions

except:
    Raised exceptions are handled, if any

else:
    Set of statements
    Executed when no exceptions are raised

finally:
    Set of statements
    Executed regardless of exceptions
```

OR we wish

Flow Diagram: Exception Handling in Python



Sanjana Menasli
244

Python Files

Def file (2M)

File is a named location on disk to store related information. It is used to permanently store data in a non-volatile memory (e.g. hard disk).

There are 2 types of file. Explain with example

File Types

Files can be categorized into two types namely, text file and binary file.

(1) Text File

Text files can only contain textual data. A text file contains several lines of text that are each followed by an End-of-Line (EOL) character. An End-of-File (EOF) marker is placed after the final character, which signals the end of the file.

Common extensions for text file formats:

- Web standards: html, xml, css, svg, json
- Source code: c, cpp, h, cs, js, py, java, php
- Documents: txt, markdown, asciidoc, rtf
- Configuration: ini, cfg, rc, reg
- Tabular data: csv, tsv

(2) Binary File

Binary files contain a sequence of bytes in various custom format types such as image, audio, and video.

Common extensions for binary file formats:

- Images: jpg, png, gif, bmp
- Videos: mp4, mkv, avi, mov, mpg
- Audio: mp3, aac, wav, wma
- Documents: pdf, doc, xls, ppt, docx
- Archive: zip, rar, 7z, tar, iso
- Database: mdb, sqlite
- Executable: exe, dll, class

File Names and File Paths

Filename refers to the name of the actual file whereas the path refers to the exact location of a file in the system. General naming conventions for an individual file are a file name and an extension, separated by a period. The path to a specified file consists of one or more components, separated by a special character (a backslash for Windows and forward slash for Linux), with each component usually being a directory name or file name, and possibly a volume name or drive name in Windows or root in Linux. If a component of a path is a file name, it must be the last component.

Deleting a file

Files can be deleted by **remove()** function which is present in the **os** module. Hence, we need to import **os** module first and call the **remove()** function by passing name of the file as an argument/parameter. If specified file does not exist, it throws `FileNotFoundError` exception.

Syntax:

```
os.remove('file_name')
```

Example:

```
>>> import os
>>> os.remove('test.txt')
```

Object Oriented Programming in Python

Classes and Objects

Define & Syntax

Class is a template or blueprint through which objects are created.

Object is an instance of a class. Object is a real-world entity which contains variables (data attributes) representing the state of the entity and the methods representing the behaviour or functionality.

Creating Classes and Objects

Syntax of creating a class:

```
class ClassName:
    data_attributes
    methods()
```

Syntax of creating an object:

```
object_name = class_name()
```

Constructor

The constructor in Python is implement using **`__init__()`** method. The **`__init__()`** method is invoked every time an object is created from a class. The **`__init__()`** method lets the class initialize the object's attributes.

The `self` Parameter

The **`self`** parameter is a reference to the current instance of the class and is used to access variables that belongs to the class. It does not have to be named **`self`**, you can call it whatever you like, but it must be the first parameter of any method in the class.

Example:

```
class mobile:
    def __init__(self):
        print('You just initialized an object')
    def receive_message(self, model):
        self.model = model
    def send_message(self):
        print('Your mobile model is', self.model)
```

```
realme = mobile()
realme.receive_message('realme 2 pro')
realme.send_message()
```

```
You just initialized an object
Your mobile model is realme 2 pro
```

Encapsulation

- Encapsulation is one of the fundamental concepts in object-oriented programming (OOP).
- Encapsulation is the process of combining variables that store data and methods that work on those variables into a single unit called class.
- Encapsulation ensures that the object's internal representation (its state and behaviour) is hidden from the rest of the application.
- Encapsulation makes the concept of data hiding possible.
- Abstraction is a process where you show only "relevant" variables that are used to access data and "hide" implementation details of an object from the user.

Classes with Multiple Objects

A class can have any number of objects.

```
class Complex:
    def __init__(self, real=0, imaginary=0):
        self.real = real
        self.imaginary = imaginary
    def add(self, c1, c2):
        self.real = c1.real+c2.real
        self.imaginary = c1.imaginary+c2.imaginary
        return self
    def show(self):
        print(self.real, '+ i', self.imaginary)
c1 = Complex(5, 10.5)
c2 = Complex(2, 3.3)
c3 = Complex()
c3.add(c1, c2)
c1.show()
c2.show()
c3.show()
```

```
5 + i 10.5
2 + i 3.3
7 + i 13.8
```

In the above program, the `add()` method accepts objects as parameters and returns an object.

```
c3.add(c1, c2)
```

The `c3` object invokes `add()` method by passing `c1` and `c2` as two objects. The `self` object in the `add()` method represents `c3`, which is returned by it.

Inheritance

- Acquiring properties of one class into another class is called inheritance.
- Properties of one class are derived into another class.
- A class that is used as the base for inheritance is called a **super class** or **base class** or **parent class**.
- A class that inherits from a base class is called a **sub class** or **derived class** or **child class**.

Syntax:

```
class derived_class_name ( base_class_name ):
        <statement 1>
        ..
        ..
        <statements N>
```

Types of Inheritance

1. Single Inheritance
2. Multilevel Inheritance
3. Multiple Inheritance
4. Hierarchical Inheritance
5. Multipath Inheritance

Single Inheritance

In this, there is one base class from which a new class is derived so the derived class inherits the properties of base class.


```
class A:
    statements...
class B(A):
    statements...
```

```
class A:
    def displayA(self):
        print('Inside base class')
class B(A):
    def displayB(self):
        print('Inside derived class')

obj = B()
obj.displayA()
obj.displayB()
```

```
Inside base class
Inside derived class
```

Multilevel Inheritance

A derived class further can be used as base class to derive another class, such an inheritance is called as multilevel inheritance.


```
class A:
    statements...
class B(A):
    statements...
class C(B):
    statements...
```

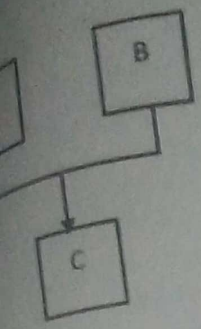
```
class A:
    def displayA(self):
        print('Inside A')
class B(A):
    def displayB(self):
        print('Inside B')
class C(B):
    def displayC(self):
        print('Inside C')

obj = C()
obj.displayA()
obj.displayB()
obj.displayC()
```

```
Inside A
Inside B
Inside C
```

Multiple Inheritance

In this type, a class is derived from multiple base classes. In C# multiple inheritance is implemented using interface.

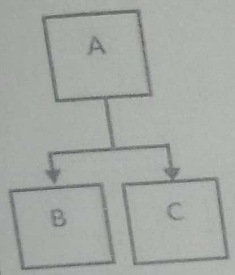
```
class A:
    def displayA(self):
        print('Inside A')
```




```
class A:
    statements...
class B:
    statements...
class C(A, B):
    statements...
```

Hierarchical Inheritance

Deriving many new classes from one base class is called hierarchical inheritance.



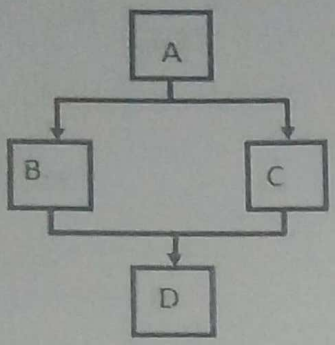

```
class A:
    statements...
class B(A):
    statements...
class C(A):
    statements...
```

```
class A:
    def displayA(self):
        print('Inside A')
class B(A):
    def displayB(self):
        print('Inside B')
class C(A):
    def displayC(self):
        print('Inside C')

objB = B()
objC = C()

objB.displayA()
objB.displayB()

objC.displayA()
objC.displayC()
```

```
Inside A
Inside B
Inside A
Inside C
```

Multipath Inheritance

A multipath inheritance is a class (D) which is derived from two classes (B and C) that are derived from a same base class (A).




```
class A:
  statements...
class B(A):
  statements...
class C(A):
  statements...
class D(B, C):
  statements...
```

```
1  class A:
2      def displayA(self):
3          print('Inside A')
4  class B(A):
5      def displayB(self):
6          print('Inside B')
7  class C(A):
8      def displayC(self):
9          print('Inside C')
10 class D(B, C):
11     def displayD(self):
12         print('Inside D')
13 obj = D()
14 obj.displayA()
15 obj.displayB()
16 obj.displayC()
17 obj.displayD()
```

```
Inside A
Inside B
Inside C
Inside D
```

```
c2 = Circle(6)
c3 = Circle(3)
print(c1<c2)
print(c2<c3)
print(c1<c3)
```

Explain :__init__() method in python?

The __init__() method in python is a special method, often referred to as a constructor. It is automatically called when an object of a class is created. This method is used to initialize the attributes of the class and set up the object with its initial state.

Eg:

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

P1 = Person("Sanjana", 25)
print(P1.name)
print(P1.age)
```

Sanjana Menarli

Python List

Creating a List

- A `list` is a data structure that holds an ordered collection of items.
- The list of items should be enclosed in square brackets – [ ].
- List is a mutable type of data structure. We can alter the list items.
- List is ordered. List items can be accessed by using index that starts from 0. Index can be negative, which starts from right towards left.
- List can contain duplicate items.
- List can contain heterogeneous items.

Syntax.
list_name = [element1, element2, element3, ---]

Examples:

```
squares = [1, 4, 9, 16, 25, 36]
student = ['Rakesh', 215, 85.5]
colours = ['red', 'blue', 'green', 'orange', 'purple']
```

Indexing

List items are indexed (subscripted). Each item of list can be accessed using an index value, which starts at 0 and increments up to n-1 where n is length of the list. Indices may also be negative numbers, to start counting from the right.

```
squares = [1, 4, 9, 16, 25, 36]
```

1	4	9	16	25	36	
0	1	2	3	4	5	← Positive index
-6	-5	-4	-3	-2	-1	← Negative index

```
squares[0] → 1      squares[3] → 16      squares[5] → 36
squares[-1] → 36    squares[-3] → 16     squares[-6] → 1
```

Slicing

We can extract a sub list from a list by providing a range of index values within a pair of square brackets.

Following are some slicing examples:

```
li = [10, 20, 30, 40, 50, 60]
```

Slicing expression	Meaning	Output
`li[1:4]`	Items from index position 1 (included) to 4 (excluded)	[20, 30, 40]

`li[:3]`	Items from the beginning to index position 3 (excluded)	[10, 20, 30]
`li[2:]`	Items from index position 2 (included) to the end	[30, 40, 50, 60]
`li[-5:-2]`	Items in reverse from fifth item (included) to second item (excluded)	[20, 30, 40]
`li[-2:]`	Items from the second-last (included) to the end	[50, 60]
`li[:-4]`	Items from beginning to fourth-last (excluded)	[10, 20]
`li[1:5:2]`	Items from index position 1 (included) to 5 (excluded) by step 2	[20, 40]
`li[::2]`	Items from beginning to end by step 2	[10, 30, 50]
`li[::-1]`	Items from last to start by step -1. [reversed list]	[60, 50, 40, 30, 20, 10]

```
1 li = [1, 5, 3, 2]
2 print(5 in li)
3 print(10 in li)
```

```
True
False
```

Operations on Lists

Concatenation

Two or more lists can be concatenated by using + operator.

```
1 l1 = [30, 10, 30]
2 l2 = [15, 25, 5]
3 print(l1+l2)
4
```

```
[30, 10, 30, 15, 25, 5]
```

Repetition

A list can be repeated for 'n' times using * operator. It is a binary operator. First operand must be a list and second operand must be integer.

```
1 li = [10, 20]
2
3 print(li*3)
```

```
[10, 20, 10, 20, 10, 20]
```

in operator

The "in" operator is used to find presence of an item in a list. If the item is present, then it returns 'True', otherwise 'False'.

```
1 li = [1, 5, 3, 2]
2 print(5 in li)
3 print(10 in li)
```

```
True
False
```

Making copies of a list

A new list can be created copying from existing list by using following two methods:

Consider `list1 = [10, 20]`

- = operator :
  - `list2 = list1`

  In this approach, the newly created list refers to the existing list. Hence, changes done to one list will reflect in another list.

- copy() method
  - `list3 = list1`

  In this approach, the newly created list copies the content of existing list. Hence, changes done to one list will not reflect in another list.

```
1 list1 = [10, 20]
2 list2 = list1           # copying using = operator
3 list3 = list1.copy()    # copying using copy() method
4 print('list 1 = ', list1)
5 print('list 2 = ', list2)
6 print('list 3 = ', list3)
```

```
list 1 =  [10, 20]
list 2 =  [10, 20]
list 3 =  [10, 20]
```

```
1 list1.append(30)    # modification done to l1 are reflected in l2 and not in l3
```

```
1 print('list 1 = ', list1)
2 print('list 2 = ', list2)
3 print('list 3 = ', list3)
```

```
list 1 =  [10, 20, 30]
list 2 =  [10, 20, 30]
list 3 =  [10, 20]
```

Built-in Function on Lists

The following built-in functions can be applied on lists.

Consider the following examples:

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

Function	Meaning	Example	Return value
`len()`	It accepts a list object as argument and returns the total number of items present in the list.	`len(l)`	5
`max()`	It accepts a list object as argument and returns the largest item. *Data type of all the items must be same.*	`max(l)`	50
`min()`	It accepts a list object as argument and returns the smallest item. *Data type of all the items must be same.*	`min(l)`	10
`sum()`	It accepts a list object as argument and returns the sum of all items. *Data type of list items must be int/float.*	`sum(l)`	150
`sorted()`	It accepts a list object as argument and returns a **new list** containing items arranged in ascending order. *Data type of list items must be same. The original list is not modified.*	`sorted(l)`	[10, 20, 30, 40, 50]
`any()`	Returns **True** if any of the Boolean values in the list is **True**, else returns **False**.	`l([True, False, False]`	True
`all()`	Returns **True** if all the Boolean values in the list are **True**, else returns **False**.	`l[True, True, True]`	True

Methods of List

The class `list` provides following methods to manipulate list and list items.

```
li = [10, 50, 30, 10]
```

Methods	Meaning	Example	Return value / List contents
index()	Returns the index of specified value (first occurrence, if many).	`li.index(30)`	2
count()	Returns the number of occurrences of an item in the list.	`li.count(10)`	2
append()	Adds a new item at the end of the list.	`li.append(20)`	`[10, 50, 30, 10, 20]`
insert()	Adds a new item at a specified index position	`li.insert(3, 40)`	`[10, 50, 30, 40, 10, 20]`
pop()	Deletes an item at a specified index position. If no index value is provided,	`li.pop(4)`	10 `[10, 50, 30, 40, 20]`

	then removes the last item from the list. pop() function returns the deleted item.	`li.pop()`	10 [10, 50, 30, 40]
remove()	Deletes a specific item from the list. This function does not return the deleted item.	`li.remove(50)`	[10, 30, 40]
extend()	Adds another list to the existing list.	`li.extend([60, 50, 20])`	[10, 30, 40, 60, 50, 20]
copy()	Copies one list into another list.	`li2 = li.copy()`	li2 will contain a copy of li [10, 30, 40, 60, 50, 20]
sort()	Arranges the items in ascending order.	`li.sort()`	[10, 20, 30, 40, 50, 60]
reverse()	Arranges the items in reverse order.	`li.reverse()`	[60, 50, 40, 30, 20, 10]
clear()	Deletes all the items in a list.	`li.clear()`	[]

Python Tuples

Creating a Tuple

- A **tuple** is a data structure that holds an ordered collection of items.
- The items should be enclosed in a pair of parenthesis – ( ).
- Tuple is an immutable type of data structure. We cannot alter the tuple items.
- Tuple is ordered. Tuple items can be accessed by using index that starts from 0. Index can be negative, which starts from right towards left.
- Tuple can contain duplicate items.
- Tuple can contain heterogeneous items.

Examples:

```
squares = (1, 4, 9, 16, 25, 36)
student = ('Rakesh', 215, 85.5)
colours = ('red', 'blue', 'green', 'orange', 'purple')
```

Indexing

Tuple items are indexed. Each item of tuple can be accessed by using an index value, which starts at 0 and increments up to n-1 where n is length of the tuple. Indices may also be negative numbers, to start counting from the right.

```
squares = (1, 4, 9, 16, 25, 36)
```

1	4	9	16	25	36	
0	1	2	3	4	5	← Positive index
-6	-5	-4	-3	-2	-1	← Negative index

```
squares[0] → 1        squares[3] → 16        squares[5] → 36
squares[-1] → 36      squares[-3] → 16       squares[-6] → 1
```

Length of a tuple can be retrieved by using python library function called '**len()**'. In the above example, **len(squares)** returns 6.

Slicing

We can extract a sub tuple from a tuple by providing a range of index values within a pair of square brackets.

Following are some slicing examples:

```
t = (10, 20, 30, 40, 50, 60)
```

Slicing expression	Meaning	Output
t[1:4]	Items from index position 1 (included) to 4 (excluded)	(20, 30, 40)
t[:3]	Items from the beginning to index position 3 (excluded)	(10, 20, 30)

t[2:]	Items from index position 2 (included) to the end	(30, 40, 50, 60)
t[-5:-2]	Items in reverse from fifth item (included) to second item (excluded)	(20, 30, 40)
t[-2:]	Items from the second-last (included) to the end	(50, 60)
t[:-4]	Items from beginning to fourth-last (excluded)	(10, 20)
t[1:5:2]	Items from index position 1 (included) to 5 (excluded) by step 2	(20, 40)
t[::2]	Items from beginning to end by step 2	(10, 30, 50)
t[::-1]	Items from last to start by step -1. [reversed tuple]	(60, 50, 40, 30, 20, 10)

Operations on Tuple

Concatenation

Two or more tuples can be concatenated by using + operator to create another tuple.

```
1 t1 = (10, 20)
2 t2 = (5, 15)
3 t3 = t1 + t2
4 print(t3)
```

```
(10, 20, 5, 15)
```

Repetition

A tuple can be repeated for 'n' times using * operator. It is a binary operator. First operand must be a tuple and second operand must be integer.

```
1 t = (10, 20)
2 print(t*3)
```

```
(10, 20, 10, 20, 10, 20)
```

The in Operator

The "in" operator is used to find presence of an item in a tuple. If the item is present, then it returns 'True', otherwise 'False'.

```
1 t = (5, 20, 15, 10)
2 print(15 in t)
3 print(30 in t)
```

```
True
False
```

Built-in Function on Tuple

The following built-in functions can be applied on tuples.

Consider the following examples:

```
t = (20, 10, 40, 50, 30)
```

Function	Meaning	Example	Return value
`len()`	It accepts a tuple object as argument and returns the total number of items present in the tuple.	`len(t)`	5
`max()`	It accepts a tuple object as argument and returns the largest item. *Data type of all the items must be same.*	`max(t)`	50
`min()`	It accepts a tuple object as argument and returns the smallest item. *Data type of all the items must be same.*	`min(t)`	10
`sum()`	It accepts a tuple object as argument and returns the sum of all items. *Data type of tuple items must be int/float.*	`sum(t)`	150
`sorted()`	It accepts a tuple object as argument and returns a **new tuple** containing items arranged in ascending order. *Data type of tuple items must be same. The original tuple is not modified.*	`sorted(t)`	[10, 20, 30, 40, 50]
`any()`	Returns **True** if any of the Boolean values in the tuple is **True**, else returns **False**.	`t([True, False, False]`	True
`all()`	Returns **True** if all the Boolean values in the tuple are **True**, else returns **False**.	`t[True, True, True]`	True

Methods of Tuple

Tuples are immutable, that means, we cannot modify the tuple items. Hence, the `tuple` class provides two methods.

```
t = (10, 50, 30, 10)
```

Methods	Meaning	Example	Return value
index()	Returns the index of specified item (first occurrence, if many).	`t.index(50)`	1
count()	Returns the number of occurrences of an item in a tuple.	`t.count(10)`	2

Python Strings

Creating and Storing Strings

Strings consist of one or more characters enclosed by quotation marks.

Define strings in python. give an example.
→ A string is a sequence of characters enclosed in single ('), double (") or triple (''') quotes. Strings are used to store & manipulate text.

```
1  s1 = "Python"
2  s2 = 'Programming'
3  s3 = 'C'
4  s4 = '''This is
5  multiline
6  string'''
7
8  print(s1)
9  print(s2)
10 print(s3)
11 print(s4)
```

```
Python
Programming
C
This is
multiline
string
```

s1 → string defined within double quotation marks
s2 → string defined within single quotation marks
s3 → string with single character
s4 → string with multiple lines

The data type of string is implemented as an object of '**str**' class in python. The output of `type(s1)` gives us `<class 'str'>`.

Accessing Sting Characters

- Each character of string can be accessed using an index value
- Index starts at 0 and increments up to n-1 where n is length of the string.
- Indices may also be negative numbers, to start counting from the right.
- Negative indexing starts with −1 index corresponding to the last character in the string and then the index decreases by one as we move to the left.

p	y	t	h	o	n
0	1	2	3	4	5
-6	-5	-4	-3	-2	-1

← Positive index
← Negative index

```
s[0] → 'p'
s[-1] → 'n'
s[3] → 'h'
s[-6] → 'p'
s[5] → 'n'
```

Strings are immutable, that means, we *cannot modify the string contents*. In the above example, if we try to update `s[2]='T'`, the interpreter throws error. However, we can re-initialize string object s to hold new value, such as, `s='Java'`.

the str() function

The str() function can be used to convert an object of one data type to string type.

In the below example, age is defined of type int, which is then converted to string using str() function.

```
1  age = 23
2  s = str(age)
3  print(type(s))
```

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

Operations on Strings

Concatenation

Two or more strings can be concatenated by using + operator. The operands of + must of string type, otherwise 'TypeError' is raised.

```
1  s1 = 'face'
2  s2 = 'book'
3  print(s1+s2)
```

```
facebook
```

Repetition

A string can be repeated for 'n' time using * operator. It is a binary operator. First operand must be string and second operand must be integer.

```
1  s='python'
2  print(s*3)
```

```
pythonpythonpython
```

in operator

The in operator is used to find presence of a substring in another string. If a substring is present in another string, then it returns 'True', otherwise 'False'.

```
1 s1 = 'hackathon'
2 s2 = 'hack'
3 s3 = 'thon'
4 s4 = 'pack'
5 print(s2 in s1)
6 print(s3 in s1)
7 print(s4 in s1)
8
```

```
True
True
False
```

Comparison

The relational operators (<, <, <=, >=, ==, !=) can be used to compare two strings, which returns 'True' or 'False'. Python compares the ASCII values of each character to compare.

```
1 s1 = 'hello'
2 s2 = 'hi'
```

```
1 print(s1==s2)
```

```
False
```

```
1 print(s1!=s2)
```

```
True
```

```
1 print(s1<s2)
```

```
True
```

(What is slicing Explain with Example. 5M/10M)

Slicing

We can extract a sub string from a string by providing a range of index values within a pair of square brackets.

Following are some slicing examples:

```
s = 'Python'
```

Slicing expression	Meaning	Output
s[1:4]	Characters from index position 1 (included) to 4 (excluded)	'yth'
s[:3]	Characters from the beginning to index position 3 (excluded)	'Pyt'
s[2:]	Characters from index position 2 (included) to the end	'thon'
s[-5:-2]	Characters in reverse from fifth character (included) to second character (excluded)	'yth'

s[-2:]	Characters from the second-last (included) to the end	'on'
s[:-4]	Characters from beginning to fourth-last (excluded)	'Py'
s[1:5:2]	Characters from index position 1 (included) to 5 (excluded) by step 2	'yh'
s[::2]	Characters from beginning to end by step 2	'pto'
s[::-1]	Characters from last to start by step -1. [reversed string]	'nohtyP'

Joining

Strings can be joined with the join() method.

Syntax:

string_name.join(sequence)

Here sequence can be string or list.

- If the sequence is a **string**, then join() function inserts *string_name* between each character of the string *sequence* and returns the **concatenated string**.

```
1 num='123'
2 s = 'python'
3 s.join(num)
```

```
'1python2python3'
```

```
1 '-'.join('hello')
```

```
'h-e-l-l-o'
```

- If the sequence is a **list**, then join() function inserts *string_name* between each item of list *sequence* and returns the **concatenated string**.

It should be noted that all the items in the list should be of string type.

```
1 dob = ['12', '04', '1999']
2 '/'.join(dob)
```

```
'12/04/1999'
```

```
1 li = ['hello', 'this', 'is', 'so', 'wonderful']
2 ' '.join(li)
```

```
'hello this is so wonderful'
```

Traversing

String is a sequence of characters, hence each of these characters can be traversed using the for loop.

```
1  s = 'python'
2  for character in s:
3      print(character)
```

```
p
y
t
h
o
n
```

```
1  s = 'python'
2  for i in range(len(s)):
3      print('index of', s[i], '=', i)
```

```
index of p = 0
index of y = 1
index of t = 2
index of h = 3
index of o = 4
index of n = 5
```

Format Specifiers

Python supports multiple ways format strings.

Using %

Various format specifiers such as %d and %f can be used to format a string.

```
1  print('cost of %d pen is %f' % (1, 5.5))
```

```
cost of 1 pen is 5.500000
```

```
1  print('cost of %d pen is %.2f' % (1, 5.5))
```

```
cost of 1 pen is 5.50
```

What are String methods? Explain the methods with example.

String Methods

The `str` class has following methods that can be applied on string objects.

```
s = 'PYTHON is programmer FRIENDLY'
```

Method	Meaning	Example	Return value
index()	Returns the index of specified value (first occurrence, if many).	`s.index('N')`	5
isalpha()	Returns True, if the string contains alphabets (only)	`s.isalpha()`	False (as string contains space)
isdigit()	Returns True, if the string contains digits (only)	`s.isdigit()`	False
isalnum()	Returns True, if the string contains alphabets/digits	`s.isalnum()`	False
islower()	Returns True, if string is in lowercase	`s.islower()`	False
isupper()	Returns True, if string is in uppercase	`s.isupper()`	False
lower()	Converts to lowercase	`s.lower()`	'python is programmer friendly'
upper()	Converts to uppercase	`s.upper()`	'PYTHON IS PROGRAMMER FRIENDLY'
title()	Converts the first character of each word to upper case	`s.title()`	'Python Is Programmer Friendly'
capitalize()	Converts the first character to upper case	`s.capitalize()`	'Python is programmer friendly'
swapcase()	Swaps cases, lower case becomes upper case and vice versa	`s.swapcase()`	'python IS PROGRAMMER friendly'
count()	Returns the number of times a specified value occurs in a string	`s.count('r')`	3
replace()	Returns a string where a specified value is replaced with a specified value	`s.replace('programmer', 'user')`	'PYTHON is user FRIENDLY'
split()	Splits the string at the specified separator, and returns a ***list***	`s.split(' ')`	['PYTHON', 'is', 'programmer', 'FRIENDLY'] *Note that a space is passed as argument to split() method.*

Explain Library functions with example.

Other Library Functions

The basic functions of python such as `len()`, `max()`, `min`, `sorted()` can also be applied to strings. Consider the following examples:

```
s = 'aAbB19'
```

Function	Meaning	Example	Return value
len()	It accepts string object as argument and returns the length of specified string.	len(s)	6
max()	It accepts string object as argument and return the largest item *(based on the ASCII value)*.	max(s)	'b'
min()	It accepts string object as argument and return the smallest item *(based on the ASCII value)*.	min(s)	'1'
sorted()	It accepts string object as argument and returns a list containing each characters of string arranged in ascending order *(based on the ASCII value)*.	sorted(s)	['1', '9', 'A', 'B', 'a', 'b']

Sanjana Menashi 244

Python Tuple

2M

Creating a Set

- Set is collection of unique items.
- Enclosed in a pair of parenthesis - { }.
- Set ***cannot*** contain duplicate items.
- Set items are unordered. We cannot access items using index.
- Set items are mutable.
- Set supports mathematic operations such as Union, Intersection, Difference, and Symmetric Difference.

Operations on Set (5M)

Union

- The union of two sets is the set of all the elements of both the sets without duplicates.
- Can be implemented using **union()** method or the | operator.

```
1 a = {10, 20, 30}
2 b = {20, 30, 40}
3 print(a | b)
4 print(a.union(b))
```

```
{20, 40, 10, 30}
{20, 40, 10, 30}
```

Intersection

- The intersection of two sets is the set of all the common elements of both the sets.
- Can be implemented using **intersection()** method ort the **&** operator.

```
1 a = {10, 20, 30}
2 b = {20, 30, 40}
3 print(a&b)
4 print(a.intersection(b))
```

```
{20, 30}
{20, 30}
```

Difference

- The difference between two sets is the set of all the elements in first set that are not present in the second set.
- Can be implemented using the **difference()** method or the – operator.

```
a = {10, 20, 30}
b = {20, 30, 40}
print('a difference b')
print(a-b)
print(a.difference(b))
print('b difference a')
print(b-a)
print(b.difference(a))
```

```
a difference b
{10}
{10}
b difference a
{40}
{40}
```

Symmetric Difference

- The symmetric difference between two sets is the set of all the elements that are either in the first set or the second set but not in both.
- Can be implemented using the **symmetric_difference()** method or the **^** operator.

```
a = {10, 20, 30}
b = {20, 30, 40}
print(a^b)
print(a.symmetric_difference(b))
```

```
{40, 10}
{40, 10}
```

Built-in Function on Set

What are built functions on set & explain the functions with example Imp for 5M or 10M

The following built-in functions can be applied on sets.

Consider the following examples:

```
s = {20, 10, 40, 50, 30}
```

Function	Meaning	Example	Return value
`len()`	It accepts a set object as argument and returns the total number of items present in the set.	`len(s)`	5
`max()`	It accepts a set object as argument and returns the largest item. *Data type of all the items must be same.*	`max(s)`	50
`min()`	It accepts a set object as argument and returns the smallest item. *Data type of all the items must be same.*	`min(s)`	10
`sum()`	It accepts a set object as argument and returns the sum of all items. *Data type of set items*	`sum(s)`	150

	must be int/float.		
`sorted()`	It accepts a set object as argument and returns a **new list** containing items arranged in ascending order. *Data type of set items must be same. The original set is not modified.*	`sorted(s)`	[10, 20, 30, 40, 50]
`any()`	Returns **True** if any of the Boolean values in the set is **True**, else returns **False**.	`s({True, False, False}`	True
`all()`	Returns **True** if all the Boolean values in the set are **True**, else returns **False**.	`S{True, True, True}`	True

Methods of Set

The class **set** provides following methods to manipulate set items.

add( )

Adds an item to the set *set_name*.

Syntax:

set_name.add(item)

Example:

```
1  s = {10, 30, 20}
2  s.add(5)
3  print(s)
```

```
{10, 20, 5, 30}
```

update( )

Update one set by adding items from another set.

Syntax:

s1.update(s2)

Example:

```
1  a = { 10, 20, 30}
2  b = {5, 15, 20, 25}
3  print('before update set a', a)
4  a.update(b)
5  print('after update set a', a)
```

```
before update set a {10, 20, 30}
after update set a {20, 5, 25, 10, 30, 15}
```

remove( )

Removes an item from the set. It raises ***KeyError*** if the item is not present in the set.

Syntax: