Course Title: PYTHON PROGRAMMING LANGUAGE AND MACHINE LEARNING IN BIOINFORMATICS	**Course Code: GNKPSBIMJ2503**
Unit 1	**Introduction to Python and OOPs Concept**
Unit 2	**Regular Expression and Pattern Matching**
Unit 3	**Biopython**
Unit 4	**Machine Learning using Numpy**

Discussion

- What you understand by Programming Language?
- Which Programming Language do you like the most and why?
- Is Learning Programming Language being essential in today's world?

UNIT I:
Introduction to Python and OOPs Concept

Differences between program and scripting language

- Program

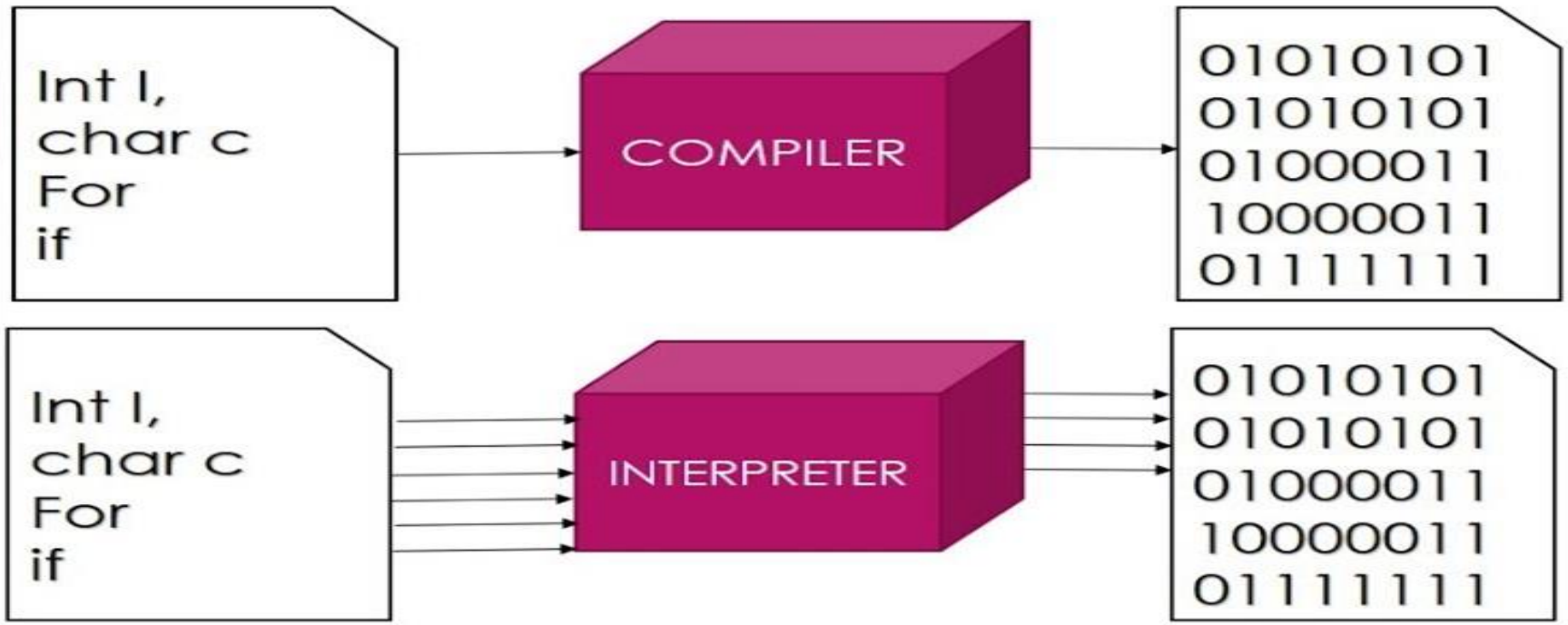
- a program is executed *(i.e. the source is first compiled, and the result of that compilation is expected)*
- A "program" in general, is a sequence of instructions written so that a computer can perform certain task. More syntax based
- All lines are checked once and errors are thrown.

- Scripting

- a script is interpreted.
- A "script" is code written in a scripting language. It mostly used to control other applications. Not more syntax based

Interpreter checks line by line errors.

SCRIPTING V/S PROGRAMMING

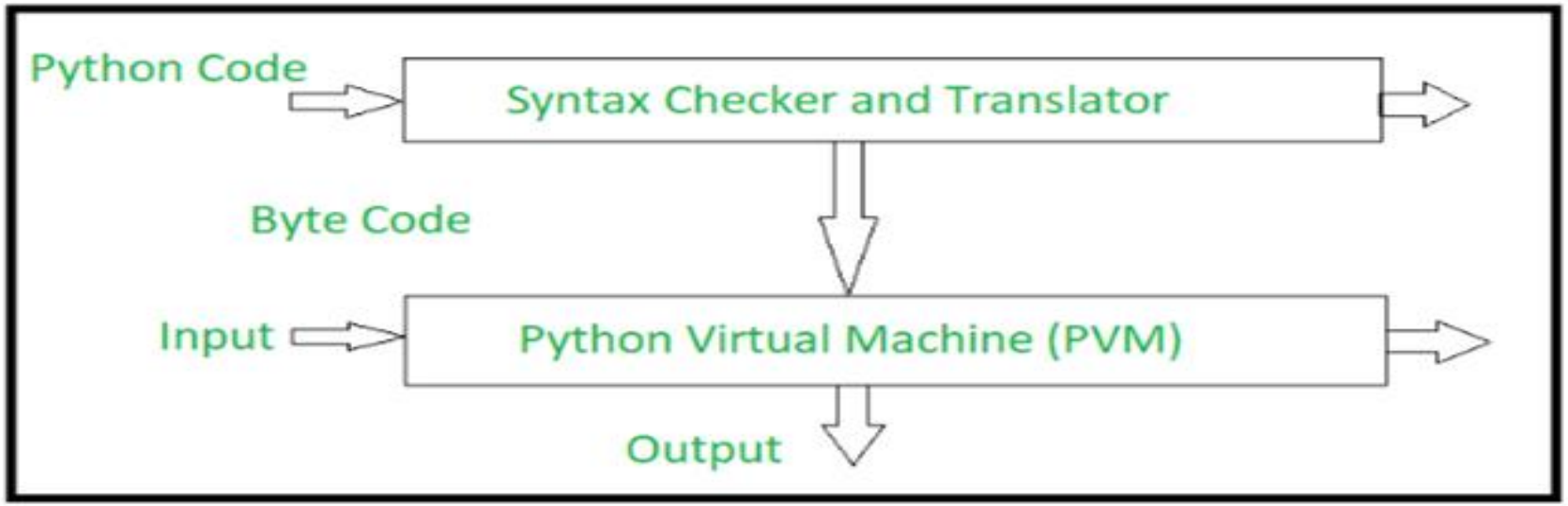


What is Python…?

- Python is a **general purpose programming language** that supports both OOPs and procedural also.
- So, Python is **programming language as well as scripting language.**
- Python is also called as **Interpreted language**

History

- Invented in the Netherlands, early 90s by **Guido van Rossum**
- Python was conceived in the late 1980s and its implementation was started in December 1989
- Guido Van Rossum is fan of 'Monty Python's Flying Circus', this is a famous TV show in Netherlands



The Python interpreter performs following tasks to execute a Python program :

- **Step 1 :** The interpreter reads a python code or instruction. Then it verifies that the instruction is well formatted, i.e. it checks the syntax of each line.If it encounters any error, it immediately halts the translation and shows an error message.
- **Step 2 :** If there is no error, i.e. if the python instruction or code is well formatted then the interpreter translates it into its equivalent form in intermediate language called "Byte code".Thus, after successful execution of Python script or code, it is completely translated into Byte code.
- **Step 3 :** Byte code is sent to the Python Virtual Machine(PVM).Here again the byte code is executed on PVM.If an error occurs during this execution then the execution is halted with an error message.

Advantages of Python

- Most programs in **Python require considerably less number of lines of code** to perform the same task compared to other languages like C .
- **Less programming errors and reduces the development time needed also.**
- Though Perl is a powerful language ,it is **highly syntax oriented, and many modules are not completed.**

Scope of Python

- Science
  - Bioinformatics
- System
- Administration
- Web Application Development
  - -CGI
  - Testing scripts

Why do people use Python…?

The following primary factors cited by Python users seem to be these:

- Python is object-oriented

  Structure supports such concepts as polymorphism, operation overloading, and multiple inheritance.

- Indentation

  Indentation is one of the greatest future in Python.

- It's free (open source)

  Downloading and installing Python is free and easy

  Source code is easily accessible

- It's powerful
  - Dynamic typing
  - Built-in types and tools
  - Library utilities
  - Third party utilities (e.g. Numeric, NumPy, SciPy)
  - Automatic memory management
- It's portable
  - Python runs virtually every major platform used today
  - As long as you have a compatible Python interpreter installed, Python programs will run in exactly the same manner, irrespective of platform.

- It's mixable

  - Python can be linked to components written in other languages easily
  - Linking to fast,compiled code is useful to computationally intensive problems
  - - Python/C integration is quite common

- It's easy to use

  - No intermediate compile and link steps as in C/ C++
  - Python programs are compiled automatically to an intermediate form called *bytecode*, which the interpreter then reads
  - This gives Python the development speed of an interpreter without
  - the performance loss inherent in purely interpreted languages

- It's easy to learn

  - Structure and syntax are pretty easy to grasp

Who uses python today…

Python is being applied in real revenue-generating products by real companies. For instance:

- Google makes extensive use of Pythonin its web search system, and employs Python's creator.
- Intel, Cisco, Hewlett-Packard, Seagate, Qualcomm, and IBM use Python for hardware testing.
- ESRI(Environmental Systems Research Institute) uses Pythonas an end-user customization tool for its popular GIS mapping products.
- The YouTube video sharing serviceis largely writtenin Python

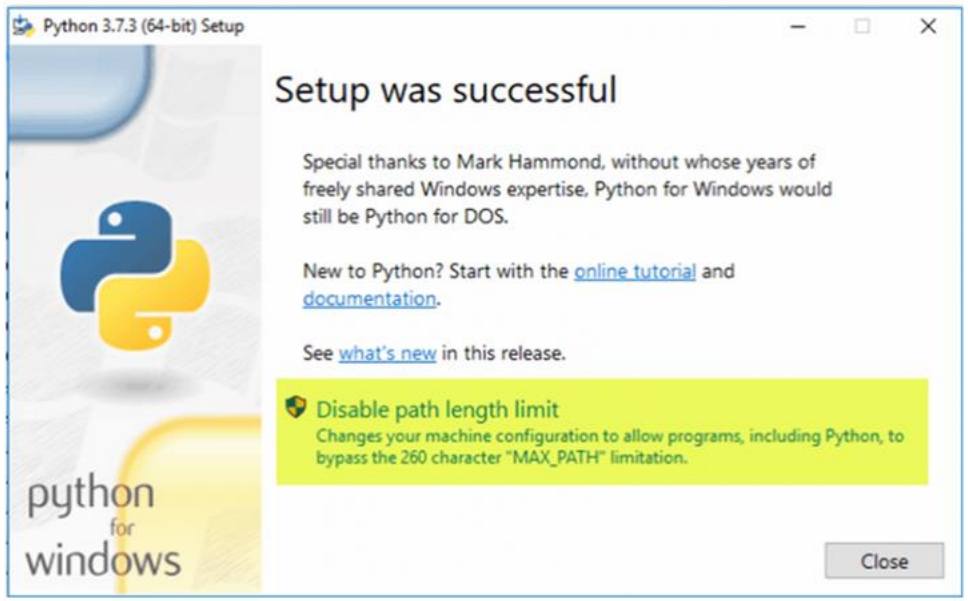
Installing Python

- Python is pre-installed on most Unix systems, including Linux and MAC OS

- But for in Windows Operating Systems , user can download from the https://www.python.org/downloads/
  - from the above link download latest version of python IDE and install, recent version is 3.4.1 but most of them uses version 2.7.7 only

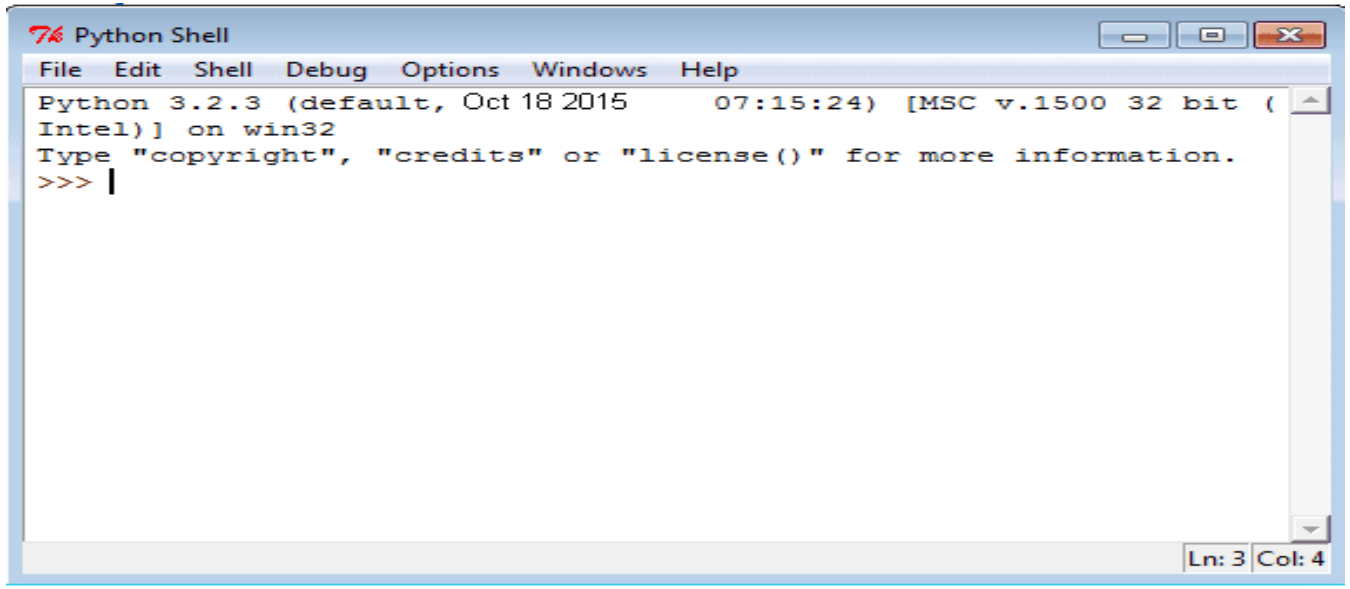
- The website appears as shown below.
- https://www.python.org/downloads/ ..


Install Python 3.7.3 (64-bit)
Select Install Now to install Python with default settings, or choose
Customize to enable or disable features.
Install Now
C:\Users\Dejan\AppData\Local\Programs\Python\Python37
Includes IDLE, pip and documentation
Creates shortcuts and file associations
Customize installation
Choose location and features
python
for
windows
Install launcher for all users (recommended)
Add Python 3.7 to PATH
Cancel


Python 3.7.3 (64-bit) Setup
Setup was successful
Special thanks to Mark Hammond, without whose years of
freely shared Windows expertise, Python for Windows would
still be Python for DOS.
New to Python? Start with the online tutorial and
documentation.
See what's new in this release.
Disable path length limit
Changes your machine configuration to allow programs, including Python, to
bypass the 260 character "MAX_PATH" limitation.
python
for
windows
Close

IDLE

- IDLE is a graphical user interface for doing Python development, and is a standard and free part of the Python system.
- It is usually referred to as an Integrated Development Environment (IDLE).
- One can write the Python code or script using the IDLE which is like an editor.This comes along with Python bundle.



Python Basics:

- Comment lines begin with: #
- File Naming Scheme
  - *filename*.py (programs)
- Example program:

```
>>  print(“Hello, World!\n”)
```

Python Data Types

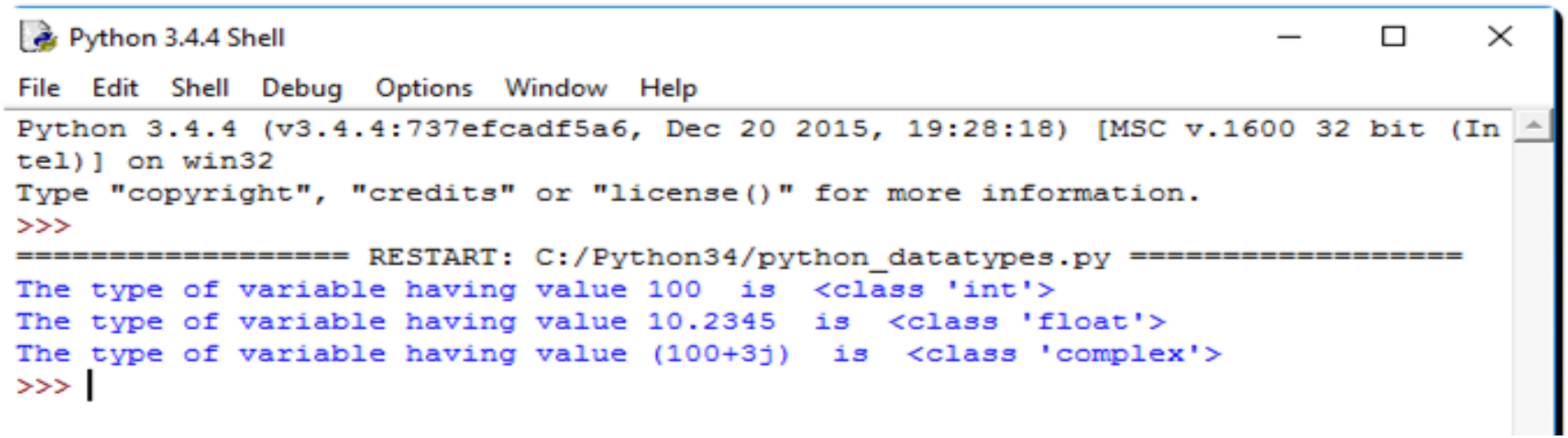
1. Python Data Type – Numeric
2. Python Data Type – String
3. Python Data Type – List
4. Python Data Type – Tuple
5. Dictionary

1. Python Data Type – Numeric

Python numeric data type is used to hold numeric values like;
1.int – holds signed integers of non-limited length.
2.long- holds long integers(exists in Python 2.x, deprecated in Python 3.x).
3.float- holds floating precision numbers and it's accurate upto 15 decimal places.
4. complex- holds complex numbers.

```
#create a variable with integer value.
a=100
print("The type of variable having value", a, " is ", type(a))

#create a variable with float value.
b=10.2345
print("The type of variable having value", b, " is ", type(b))

#create a variable with complex value.
c=100+3j
print("The type of variable having value", c, " is ", type(c))
```

If you run the above code you will see output like the below image.



Taking input in Python

input (prompt):

This function first takes the input from the user and then evaluates the expression, which means Python automatically identifies whether user entered a string or a number or list.

```
# Python program showing
val = input("Enter your value: ")
print(val)
```

Output:

```
Enter your value: 123
123
>>>
```

Basic Operators

1. Athematic Operators:

- For example ,if x= 7,y=2
- addition: x+y=9
- subtraction:x-y=5
- Multiplication:x*y=14
- Division: x/y=3.5
- Floor division: x // y=3(rounds off the answer to the nearest whole number)
- Modulus: x%y= 1(Gives the remainder when 7 is divided by 2
- Exponent : x**y=49(7 to the power of 2)

2. Relational Operators:

Relational operators compares the values. It either returns True or False according to the condition.

OPERATOR	DESCRIPTION	SYNTAX
>	Greater than: True if left operand is greater than the right	x > y
<	Less than: True if left operand is less than the right	x < y
==	Equal to: True if both operands are equal	x == y
!=	Not equal to - True if operands are not equal	x != y
>=	Greater than or equal to: True if left operand is greater than or equal to the right	x >= y
<=	Less than or equal to: True if left operand is less than or equal to the right	x <= y

```
# Examples of Relational Operators
a = 13
b = 33

print(a > b)
print(a < b)
print(a == b)
print(a != b)
print(a >= b)
print(a <= b)
```

Output:

```
False
True
False
True
False
True
```

3. **Logical operators:** Logical operators perform **Logical AND**, **Logical OR** and **Logical NOT** operations.

OPERATOR	DESCRIPTION	SYNTAX
and	Logical AND: True if both the operands are true	x and y
or	Logical OR: True if either of the operands is true	x or y
not	Logical NOT: True if operand is false	not x

 # Examples of Logical Operator

```
a = True
b = False

# Print a and b is False
print(a and b)

# Print a or b is True
print(a or b)

# Print not a is False
print(not a)
```

Output:

```
False
True
False
```

Python Membership Operators

Python's membership operators test for membership in a sequence, such as strings, lists, or tuples. There are two membership operators as explained below −

[Show Example]

Operator	Description	Example
in	Evaluates to true if it finds a variable in the specified sequence and false otherwise.	x in y, here in results in a 1 if x is a member of sequence y.
not in	Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.	x not in y, here not in results in a 1 if x is not a member of sequence y.

Example:

```
a = 10
b = 20
list = [1, 2, 3, 4, 5 ];
if ( a in list ):
   print "Line 1 - a is available in the given list"
else:
   print "Line 1 - a is not available in the given list"

if ( b not in list ):
   print "Line 2 - b is not available in the given list"
else:
   print "Line 2 - b is available in the given list"
a = 2
if ( a in list ):
   print "Line 3 - a is available in the given list"
else:
   print "Line 3 - a is not available in the given list“
```

When you execute the above program it produces the following result –

```
Line 1 - a is not available in the given list
Line 2 - b is not available in the given list
Line 3 - a is available in the given list
```

4. Assignment operators:

Assignment operators are used to assign values to the variables.

Syntax:

=

Example:

X=10

Print(x)

2. Python Data Type – String

- The string is a sequence of characters.
- Python supports Unicode characters.
- Generally, strings are represented by either single or double quotes.

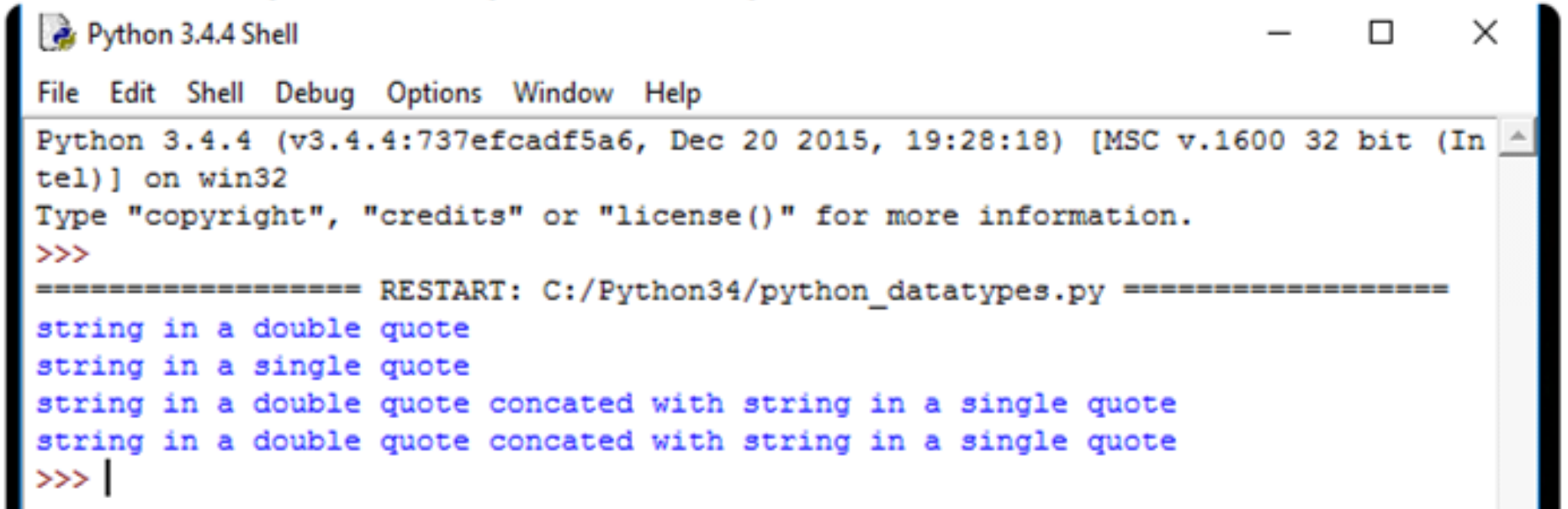
String Special Operators

Assume string variable **a** holds 'Hello' and variable **b** holds 'Python', then −

Operator	Description	Example
+	Concatenation - Adds values on either side of the operator	a + b will give HelloPython
*	Repetition - Creates new strings, concatenating multiple copies of the same string	a*2 will give -HelloHello
[]	Slice - Gives the character from the given index	a[1] will give e
[:]	Range Slice - Gives the characters from the given range	a[1:4] will give ell
in	Membership - Returns true if a character exists in the given string	H in a will give 1
not in	Membership - Returns true if a character does not exist in the given string	M not in a will give 1

```
a = "string in a double quote"
b= 'string in a single quote'
print(a)
print(b)

# using ',' to concatenate the two or several strings
print(a,"concatenated with",b)

#using '+' to concate the two or several strings
print(a+" concated with "+b)
```

The above code produce output like below picture-



Example 2:

```
str = 'Hello World!'
print str          # Prints complete string
print str[0]       # Prints first character of the string
print str[2:5]     # Prints characters starting from 3rd to 5th
print str[2:]      # Prints string starting from 3rd character
print str * 2      # Prints string two times
print str + "TEST" # Prints concatenated string
```

This will produce the following result –

```
Hello World!
H
llo
llo World!
Hello World!Hello World!
Hello World!TEST
```

Python Data Type – List

- The list is a versatile data type exclusive in Python.
- In a sense, it is the **same as the array in C/C++.**
- But the interesting thing about the list in Python is it can **simultaneously hold different types of data.**
- Formally list is an **ordered sequence** of some data written **using square brackets([]) and commas(,).**

Eg:

userAge=[15,25,30,45,50.65.55,100]

Add and Remove items

- To **add items the append() function** is useful.
- For ex : userAge.append(100) will add the value 100 to the List at the end. Now the new List is userAge=[15,25,30,45,50.65.55,100]

- To **remove** any value from the List we write **del listName[index of item to be deleted]**
- For ex: del userAge[2] will give a new List userAge=[15,25,45,50,65,55,100]

Example 1:

```
list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
tinylist = [123, 'john']

print list          # Prints complete list
print list[0]       # Prints first element of the list
print list[1:3]     # Prints elements starting from 2nd till 3rd index
print list[2:]      # Prints elements starting from 3rd element
print tinylist * 2  # Prints list two times
print list + tinylist # Prints concatenated lists
```

This produce the following result –

```
['abcd', 786, 2.23, 'john', 70.2]
abcd
[786, 2.23]
[2.23, 'john', 70.2]
[123, 'john', 123, 'john']
['abcd', 786, 2.23, 'john', 70.2, 123, 'john']
```

Updating Lists and Adding new elements:

You can update single or multiple elements of lists by giving the slice on the left-hand side of the assignment operator, and you can add to elements in a list with the append() method. For example −

Example:

```
list = ['physics', 'chemistry', 1997, 2000];
print "Value available at index 2 : “+list[2]
list[2] = 2001;
print "New value available at index 2 : “+list[2]
list.append(1998)
print  list
```

OUTPUT–

```
Value available at index 2 :1997
New value available at index 2 :
2001
'physics', 'chemistry’, 2001, 2000,1998
```

Delete List Elements

To remove a list element, you can use either the del statement if you know exactly which element(s) you are deleting or the remove() method if you do not know. For example −

Example:

```
list1 = ['physics', 'chemistry', 1997, 2000];
print list1
del list1[2];
print "After deleting value at index 2 : "
print list1
```

OUTPUT–

```
['physics', 'chemistry', 1997, 2000]
After deleting value at index 2 :
['physics', 'chemistry', 2000]
```

Built-in List Functions & Methods

Python includes the following list functions −

Sr.No.	Function with Description
1	cmp(list1, list2) Compares elements of both lists.
2	len(list) Gives the total length of the list.
3	max(list) Returns item from the list with max value.
4	min(list) Returns item from the list with min value.

Q.1. Write a Python script to store roll number of 5 students in list and perform the following:

1. Find length of a list
2. Display the smallest element from a list
3. Display largest element from a list.

1	list.append(obj) Appends object obj to list
2	list.count(obj) Returns count of how many times obj occurs in list
3	list.extend(seq) Appends the contents of seq to list
4	list.index(obj) Returns the lowest index in list that obj appears
5	list.insert(index, obj) Inserts object obj into list at offset index
6	list.pop(obj=list[-1]) Removes and returns last object or obj from list
7	list.remove(obj) Removes object obj from list
8	list.reverse() Reverses objects of list in place
9	list.sort([func]) Sorts objects of list, use compare func if given

Q.1) Write a Python script to store dna sequence a in list and perform the following:

1. Find count of occurrences of alphabet ‘atgc’ in a list
2. Display the first element from a list
3. Display remove last element from a list.

Q.2) Write a Python to store 5 number using list and display reverse of a list and sorting of a list

Python Data Type – Tuple

Tuple is another data type which is a sequence of data similar to list.

But it is **immutable**. That means data in a tuple is write protected.

Data in a tuple is written using parenthesis and commas.

Basic Tuples Operations

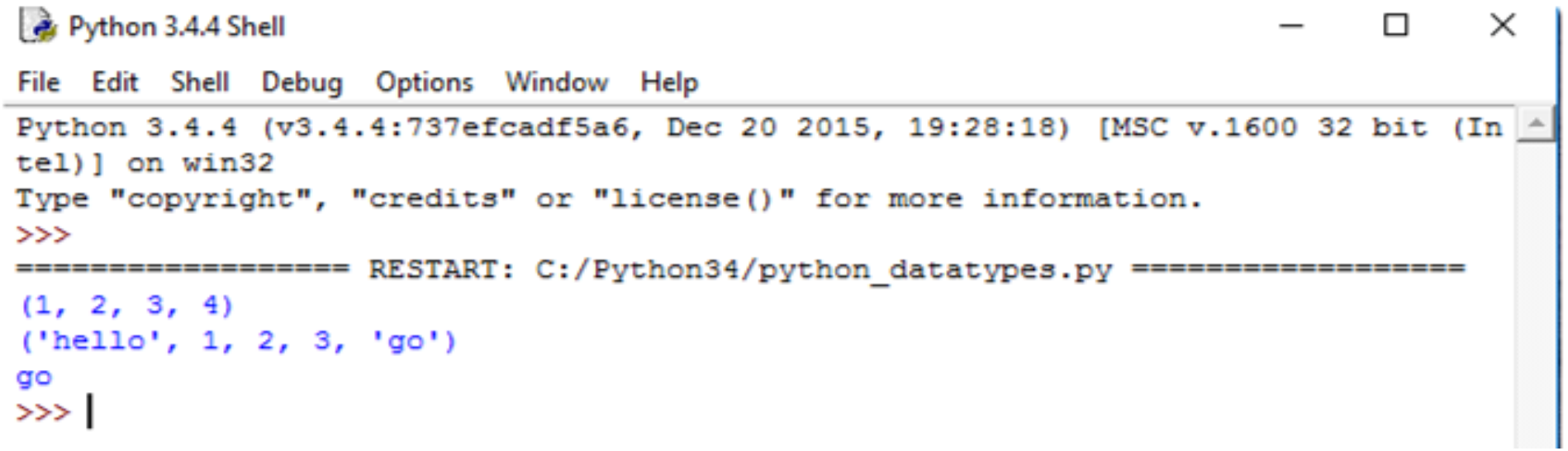
Tuples respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the result is a new tuple, not a string.

In fact, tuples respond to all of the general sequence operations we used on strings in the prior chapter −

Python Expression	Results	Description
len((1, 2, 3))	3	Length
(1, 2, 3) + (4, 5, 6)	(1, 2, 3, 4, 5, 6)	Concatenation
('Hi!',) * 4	('Hi!', 'Hi!', 'Hi!', 'Hi!')	Repetition
3 in (1, 2, 3)	True	Membership
for x in (1, 2, 3): print x,	1 2 3	Iteration

```
#tuple having only integer type of data.
a=(1,2,3,4)
print(a) #prints the whole tuple

#tuple having multiple type of data.
b=("hello", 1,2,3,"go")
print(b) #prints the whole tuple

#index of tuples are also 0 based.

print(b[4]) #this prints a single element in a tuple, in this case "go"
```

The output of this above python data type tuple example code will be like below image.



Example 2:

```
tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  )
tinytuple = (123, 'john')

print tuple               # Prints complete list
print tuple[0]            # Prints first element of the list
print tuple[1:3]          # Prints elements starting from 2nd till 3rd
print tuple[2:]           # Prints elements starting from 3rd element
print tinytuple * 2       # Prints list two times
print tuple + tinytuple # Prints concatenated lists
```

This produce the following result –

```
('abcd', 786, 2.23, 'john', 70.2)
abcd
(786, 2.23)
(2.23, 'john', 70.2)
(123, 'john', 123, 'john')
('abcd', 786, 2.23, 'john', 70.2, 123, 'john')
```

Built-in Tuple Functions

Python includes the following tuple functions −

Sr.No.	Function with Description
1	cmp(tuple1, tuple2) Compares elements of both tuples.
2	len(tuple) Gives the total length of the tuple.
3	max(tuple) Returns item from the tuple with max value.
4	min(tuple) Returns item from the tuple with min value.

Write a Python script to store roll number of 5 students in tuple and perform the following:

1. Find length of a tuple
2. Display the smallest element from a tuple
3. Display largest element from a tuple.

Dictionary

- Python Dictionary is an unordered sequence of data of key-value pair form.
- It is similar to the hash table type.
- Dictionaries are written within curly braces in the form key:value.
- It is very useful to retrieve data in an optimized way among a large amount of data.

Example :

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
```

Accessing Values in Dictionary

To access dictionary elements, you can use the familiar square brackets along with the key to obtain its value. Following is a simple example –

Example :

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
print "dict['Name']: ", dict['Name']
print "dict['Age']: ", dict['Age']
```

When the above code is executed, it produces the following

Result –

```
dict['Name']:  Zara
dict['Age']:  7
```

Updating or Adding new element to a Dictionary

You can update a dictionary by adding a new entry or a key-value pair, modifying an existing entry, or deleting an existing entry as shown below in the simple example –

Example:

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
dict['Age'] = 8; # update existing entry
dict['School'] = "DPS School"; # Add new entry

print "dict['Age']: ", dict['Age']
print "dict['School']: ", dict['School']
```

When the above code is executed, it produces the following

Result –

```
dict['Age']:  8
dict['School']:  DPS School
```

Delete Dictionary Elements

You can either remove individual dictionary elements or clear the entire contents of a dictionary. You can also delete entire dictionary in a single operation.

To explicitly remove an entire dictionary, just use the del statement. Following is a simple example −

Example:

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
del dict['Name']; # remove entry with key 'Name'
dict.clear();     # remove all entries in dict
print "dict['Age']: ", dict['Age']
print "dict['School']: ", dict['School']
```

Result:

```
dict['Age']:
Traceback (most recent call last):
  File "test.py", line 4, in <module>
    print "dict['Age']: ", dict['Age'];
TypeError: 'type' object is unsubscriptable
```

Example :

```
dict = {}
dict['one'] = "This is one"
dict[2]     = "This is two"

tinydict = {'name': 'john','code':6734, 'dept': 'sales'}

print dict['one']       # Prints value for 'one' key
print dict[2]           # Prints value for 2 key
print tinydict          # Prints complete dictionary
print tinydict.keys()   # Prints all the keys
print tinydict.values() # Prints all the values
```

This produce the following result –

```
This is one
This is two
{'dept': 'sales', 'code': 6734, 'name': 'john'}
['dept', 'code', 'name']
['sales', 6734, 'john']
```

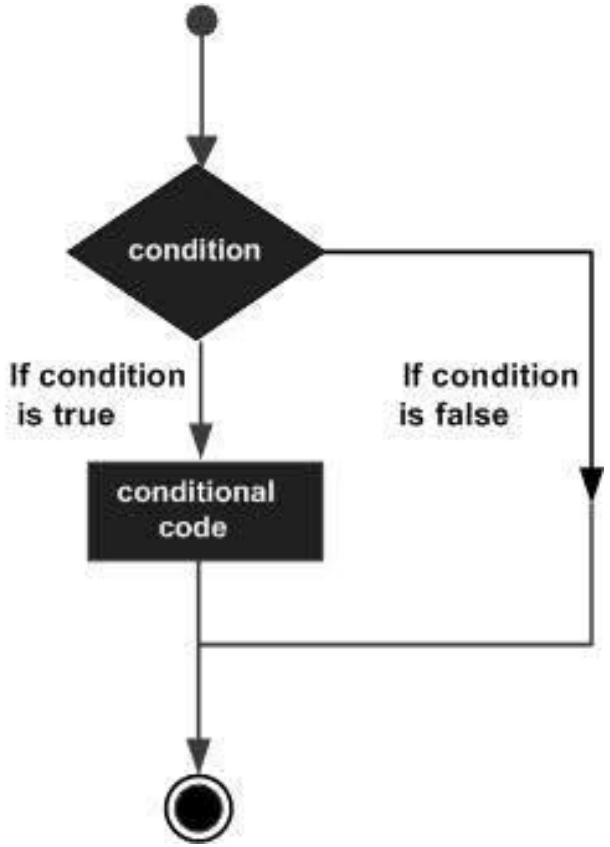
Here is the list of all Dictionary Methods

Method	Description	Syntax
copy()	Copy the entire dictionary to new dictionary	dict.copy()
update()	Update a dictionary by adding a new entry or a key-value pair to anexisting entry or by deleting an existing entry.	Dict.update([other])
sort()	You can sort the elements	dictionary.sort()
len()	Gives the number of pairs in the dictionary.	len(dict)
cmp()	Compare the values and keys of two dictionaries	cmp(dict1, dict2)
Str()	Make a dictionary into a printable string format	Str(dict)

CONDITIONAL STATEMENTS

Python - Decision Making

- Decision structures evaluate multiple expressions which produce TRUE or FALSE as outcome.
- You need to determine which action to take and which statements to execute if outcome is TRUE or FALSE otherwise.
- Following is the general form of a typical decision making structure found in most of the programming languages −

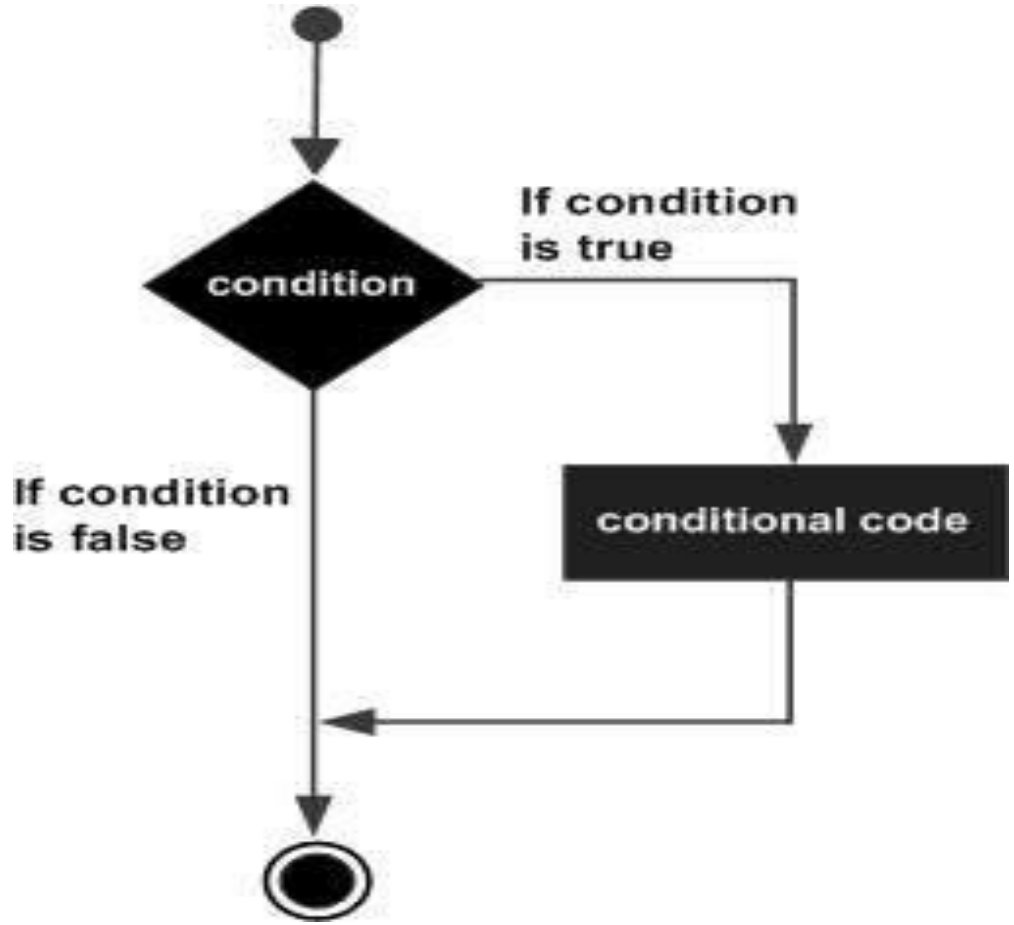


1. Python IF Statement

It is similar to that of other languages.
The if statement contains a logical expression using which data is compared and a decision is made based on the result of the comparison.

Syntax
if expression:
  statement(s)

If the boolean expression evaluates to TRUE, then the block of statement(s) inside the if statement is executed. If boolean expression evaluates to FALSE, then the first set of code after the end of the if statement(s) is executed.


condition
If condition
is true
If condition
is false
conditional code

Example:

```
var1 = 100
if var1==100:
   print "Got a true expression value"
   print var1
```

OUTPUT:

```
Got a true expression value
100
```

2. Python IF…ELSE Statements

An else statement can be combined with an if statement.
An else statement contains the block of code that executes if the conditional expression in the if statement resolves to 0 or a FALSE value.
The else statement is an optional statement and there could be at most only one else statement following if.

Syntax
The syntax of the if...else statement is −

```
if expression:
   statement(s)
else:
   statement(s)
```

Example:

```
var1 = 100
if var1==10:
   print("Got a true expression value“)
   print(var1)
else:
   print("Got a false expression value“)
   print(var1)
```

OUTPUT:

```
Got a FALSE expression value
100
```

3. Python nested IF statements

There may be a situation when you want to check for another condition after a condition resol
you can use the nested if construct.

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

Syntax

The syntax of the nested if...elif...else construct may be −

```
if expression2:
      statement(s)
   elif expression3:
      statement(s)
   elif expression4:
      statement(s)
   else:
      statement(s)
```

Example

```
var = 100
if var < 200:
   print "Expression value is less than 200"
   if var == 150:
      print "Which is 150"
   elif var == 100:
      print "Which is 100"
   elif var == 50:
      print "Which is 50"
   elif var < 50:
      print "Expression value is less than 50"
else:
   print "Could not find true expression"
```

Output−

```
Expression value is less than 200
Which is 100
```

Python Loops

Python has two primitive loop commands:

1. for loops
2. while loops

Python For Loops

A for loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).

Example

Print each fruit in a fruit list:

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

```
C:\Users\My Name>python demo_for.py
apple
banana
cherry
```

The while Loop

With the while loop we can execute a set of statements as long as a condition is true.

Example

//Print i as long as i is less than 6:

```
i = 1
while i < 6:
  print(i)
  i += 1
```

output:

12345

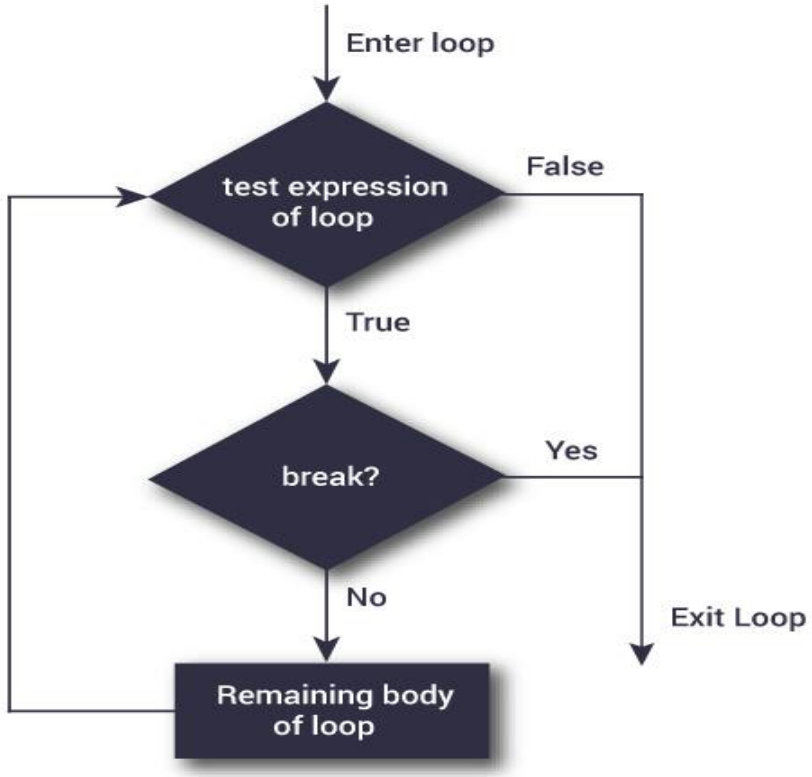
Python break statement

- The break statement terminates the loop containing it.
- Control of the program flows to the statement immediately after the body of the loop.
- If break statement is inside a nested loop (loop inside another loop), break will terminate the innermost loop and comes to outerloop.

Syntax of break

```
break
```

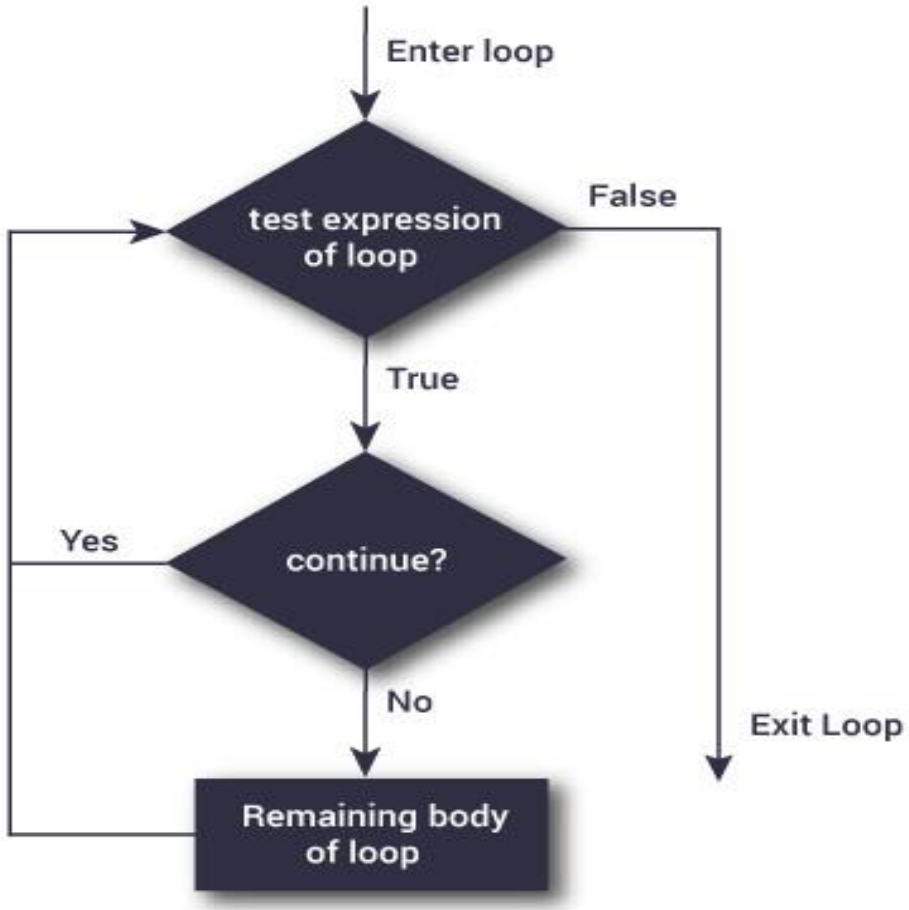
Python break statement




```
for var in sequence:
    # codes inside for loop
    if  condition:
        break
    # codes inside for loop

# codes outside for loop
```

```
while test expression:
    # codes inside while loop
    if  condition:
        break
    # codes inside while loop

# codes outside while loop
```

Continue:




```
for var in sequence:
    # codes inside for loop
    if  condition:
        continue
    # codes inside for loop

# codes outside for loop
```

```
while test expression:
    # codes inside while loop
    if  condition:
        continue
    # codes  inside while loop

# codes outside while loop
```

- Write a python script to display shipping cost based on country

  For US , <= 50 |$50

  <= 100 |$25

  <= 150 | $5 else free shipping

  For AU, <= 50 |$100 else free shipping

- Write a python script to create a list for students marks and display the count how many scored more than 60 marks.

FUNCTIONS

Python - Functions

- A function is a **block of organized, reusable code** that is used to perform a single, related action. ‘
- Functions **provide better modularity for your application** and a high degree of code reusing.
- **As you already know, Python gives you many built-in functions** like print(), etc. but you can also create your own functions. These functions are called user-defined functions.

Defining a Function

You can define functions to provide the required functionality. Here are simple rules to define a function in Python.

1. Function blocks begin with the **keyword def** followed by the **function name and parentheses** ( ) .

1. **Any input parameters or arguments should be placed within these parentheses**.

1. The code block within every function starts with a **colon (:)**.

1. The **statement return [expression] exits a function**

Syntax

```
def functionname( parameters ):
   function_suite
   return [expression]
```

Example 1:

 # Function definition is here

```
def printme( str ):
   print str
   return;
```

 # Now you can call printme function

```
printme("I'm first call to user defined function!")
printme("Again second call to the same function")
```

Output:

```
I'm first call to user defined function!
Again second call to the same function
```

Example 2:

```
# Function definition is here
def sum( arg1, arg2 ):
   total = arg1 + arg2
   print "Inside the function : ", total
return;

# Now you can call sum function

total = sum( 10, 20 );
```

1. Write a Python function that accepts a sequence using string and calculate the number of a,t,g,c.
2. Write a Python function to multiply all the numbers in a list
Sample List : (8, 2, 3, -1, 7)
Expected Output : -336

File Handling

File handling

- File handling is an important part of any web application or biological sequence.

- Python has several functions for creating, reading, updating, and deleting files.
- The key function for working with files in Python is the open() function.

- The open() function takes two parameters; filename, and mode.

- There are four different methods (modes) for opening a file:

- "r" - Read - Default value. Opens a file for reading, error if the file does not exist

- "a" - Append - Opens a file for appending, creates the file if it does not exist

- "w" - Write - Opens a file for writing, creates the file if it does not exist

- "x" - Create - Creates the specified file, returns an error if the file exists

In addition you can specify if the file should be handled as binary or text mode

1. "t" - Text - Default value. Text mode

1. "b" - Binary - Binary mode

Syntax

f = open("demofile.txt")
The code above is the same as:

f = open("demofile.txt", "r")

Note: Make sure the file exists, or else you will get an error.

read():

dnafile.txt

atgcaaatgcc
attgc
aattttgggccc

To open the file, use the built-in open() function.

Example: Read dna sequence and display

```
f = open("dnafile.txt", "r")
print(f.read())
f.close()
```

Output:

atgcaaatgcc
attgc
aattttgggccc

Note :
f.read(5): reads only 5 characters from start

readLine():

dnafile.txt
atgcaaatgcc
attgc
aattttgggccc

Example: Read first dna sequence and display

```
f = open("dnafile.txt", "r")
print(f.readLine())
f.close()
```

Output:
atgcaaatgcc

Example
Loop through the file line by line:

```
f = open("demofile.txt", "r")
for x in f:
  print(x)
f.close()
```

Write :

Write to an Existing File

To write to an existing file, you must add a parameter to the open() function:

"a" - Append - will append to the end of the file

"w" - Write - will overwrite any existing content

Example:

```
f = open("demofile2.txt", "a")
f.write(“atttgggatc")
f.close()
#open and read the file after the appending:
f = open("dnafile.txt", "r")
print(f.read())
f.close()
```

Output:

```
atgcaaatgcc
attgc
aattttgggccc
atttgggatc
```

Python Delete File

Delete a File

To delete a file, you must import the OS module, and run its os.remove() function:

Example

Remove the file "demofile.txt":

```
import os
os.remove("dnafile.txt")
```

1. Write a Python program to count the number of lines in a text file. classwork
2. Write a Python program to count the frequency of ‘at’ in a file. homework

Classwork

1. Python Program to Calculate the Area of a Triangle
2. Python Program to Find the Largest Among Three Numbers

Homework

3. Python Program to Remove Punctuations From a String
4. Write a Python program to get the 4th element and 4th element from last of a tuple.

Ouput: 6,7,9,8,7,5,5,3,2,5,6

8 and 3

Object Oriented Programming in Python

Session -3

Contents

Difference between Procedure Oriented and Object Oriented Programming

- Procedural programming **creates a step by step program** that guides the application through a sequence of instructions. Each instruction is executed in order.
- Procedural programming also **focuses on the idea that all algorithms are executed with functions and data** that the programmer has access to and is able to change.
- Object-Oriented programming is much **more similar to the way the real world works;** it is analogous to the human brain. Each program is made up of many entities called objects.
- Instead, **a message must be sent requesting the data**, just like people must ask one another for information; we cannot see inside each other's heads.

Features of OOP

- **Ability to simulate real-world event** much more effectively
- **Code is reusable** thus less code may have to be written
- **Better able to create GUI** (graphical user interface) applications
- Programmers are able to **produce faster, more accurate and better-written applications**

Fundamental concepts of OOP in Python

The four major principles of object orientation are:

- Encapsulation
- Data Abstraction
- Inheritance
- Polymorphism

What is an Object..?

- Objects are the basic run-time entities in an object-oriented system.
- They may represent a person, a place, a bank account, a table of data or any item that the program must handle.
- When a program is executed the objects interact by sending messages to one another.
- Objects have two components:
  - Data (i.e., attributes)
  - Behaviors (i.e., methods)

Object Attributes and Methods Example

Object Attributes

- Store the data for that object
- Example (taxi):
  - Driver
  - OnDuty
  - NumPassengers
  - Location

Object Methods

- Define the behaviors for the object
- Example (taxi):
  - PickUp
  - DropOff
  - GoOnDuty
  - GoOffDuty
  - GetDriver
  - SetDriver
  - GetNumPassengers

What is a Class..?

- A class is a special data type which defines how to build a certain kind of object.
- The class also stores some data items that are shared by all the instances of this class
- Instances are objects that are created which follow the definition given inside of the class
- Python doesn’t use separate class interface definitions as in some languages
- You just define the class and then use it

Methods in Classes

- Define a method in a class by including function definitions within the scope of the class block
- There must be a special first argument **self** in all of method definitions which gets bound to the calling instance
- There is usually a special method called **__init__** in most classes

A Simple Class def: Student

```
class Student:
"""A class representing a student """
def __init__(self , n, a):
    self.full_name = n
    self.age = a
def get_age(self):        #Method
    return self.age
```

- Define class:
  - Class name, begin with capital letter, by convention
  - object: class based on
- Define a method
  - Like defining a function
  - Must **have a special first parameter**, self, which provides way for a method to refer to object itself

Instantiating Objects with ‘__init__’

- __init__is the default constructor
- __init__serves as a constructor for the class. Usually does some initialization work
- An__init__method can take any number of arguments
- However, the first argument self in the definition of __init__is special

Self

- The first argument of every method is a reference to the current instance of the class
- By convention, we name this argument **self**
- In init , self refers to the object currently being created; so, in other class methods, it refers to the instance whose method was called
- Similar to the keyword **this** in Java or C++
- But Python uses self more often than Java uses this
- You **do not** give a value for this parameter(self) when you call the method, Python will provide it.

Continue…

…*Continue*

- Although you must specify self explicitly when defining the method, you don't include it when calling the method.
- Python passes it for you automatically

Defining a method:

(this code inside a class definition.)

```
def get_age(self, num):
    self.age = num
```

Calling a method:

```
>>> x.get_age(23)
```

Deleting instances: No Need to “free”

- When you are done with an object, you don’t have to delete or free it explicitly.
- Python has automatic garbage collection.
- Python will automatically detect when all of the references to a piece of memory have gone out of scope. Automatically frees that memory.
- Generally works well, few memory leaks
- There’s also no “destructor” method for classes

Syntax for accessing attributes and methods

```
>>> f = student("Python", 14)

>>> f.full_name # Access attribute
"Python"

>>> f.get_age() # Access a method
14
```

Classwork:

1. Write a python program to create class student and display roll_no, age, name and marks with only default constructor. 3
2. Write a python program to create class Sequence and store rna and dna sequence in string and display the same with only default constructor. 6

USER INPUT:

There are cases when we need to get user input in a python program then use it during the initialization of a class. For example, if we have a class 'person' that has two attributes first name and last name.

```
class Person:
def __init__(self, first_name, last_name):
        self.firstname = first_name
        self.lastname = last_name
    def show_full_name(self):
        return self.firstname + ' ' + self.lastname
```

To create a person object using this class, we could supply arguments directly requiring user input. See below.

```
F= input("enter first name")
L= input("enter last name")
person1 = Person(F,L)
person1.show_full_name()
```

Encapsulation

- Important advantage of OOP consists in the **encapsulation of data.** We can say that object-oriented programming relies heavily on encapsulation.
- **The terms encapsulation and abstraction (also data hiding)** are often used as synonyms.
- Generally speaking, encapsulation is the mechanism for restricting the access to some of an objects's components, this means, that the internal representation of an **object can't be seen from outside of the objects definition.**

- C++, Java, and C# rely on the **public, private, and protected keywords** in order to implement variable scoping and encapsulation
- It's nearly always possible to circumvent this protection mechanism

Public, Protected and Private Data

- If an identifier doesn't start with an underscore character "_" **it can be accessed from outside, i.e. the value can be read and changed**
- Data can be protected by making members private or protected. Instance variable names starting with **two underscore characters cannot be accessed from outside of the class.**

- If an identifier is only preceded by one underscore character, it is a protected member.
- Protected members can be accessed like public members from outside of class(only inherited class)

Example:

```
class Encapsulation(object):
        def__init__(self, a, b, c):
                self.public = a
                self._protected = b
                self.__private = c
```

The following interactive sessions shows the behavior of public, protected and private members:

```
>>> x = Encapsulation(11,13,17)
>>> x.public
 11
>>> x._protected
13
>>> x._protected = 23
>>> x._protected
23
>>> x.__private
    Traceback (most recent call last):
          File "<stdin>", line 1, in <module>
    AttributeError: 'Encapsulation' object has no attribute '__private‘
 >>>
```

The following table shows the different behavior Public, Protected and Private Data

Name	Notation	Behavior
name	Public	Can be accessed from inside and outside
_name	Protected	Like a public member, but they shouldn't be directly accessed from outside only inherited class can access the members
__name	Private	Can't be seen and accessed from outside

Encapsulation Using Private Members

```
class Rectangle:
        __length = 0 #private variable
        __breadth = 0#private variable
      def __init__(self): #constructor
            self.__length = 5
            self.__breadth = 3 #printing values of the private variable within the class
           print(self.__length)
           print(self.__breadth)

rec = Rectangle() #object created for the class 'Rectangle’

#printing values of the private variable outside the class using the object created for the class
'Rectangle’

print(rec.length)
print(rec.breadth)
```

Output:

```
5
3
Traceback (most recent call last):
File "main.py", line 11, in <module>
print(rec.__length)
AttributeError: 'Rectangle' object has no attribute '__length'
```

Encapsulation Using PROTECTED Members

Example:

```
class Shape:#protected variables
    _length = 10
  _breadth = 20

class Circle(Shape):
        def __init__(self): #printing protected variables in the derived class
            print(self._length)
            print(self._breadth)

cr = Circle()
#printing protected variablesoutsidethe class 'Shape' in which they are defined
print(cr._length)
print(cr._breadth)
```

Output:

```
10
20
Traceback (most recent call last):
File "main.py", line 11, in <module>
print(cr.length)
AttributeError: 'Circle' object has no attribute 'length'
```

Classwork

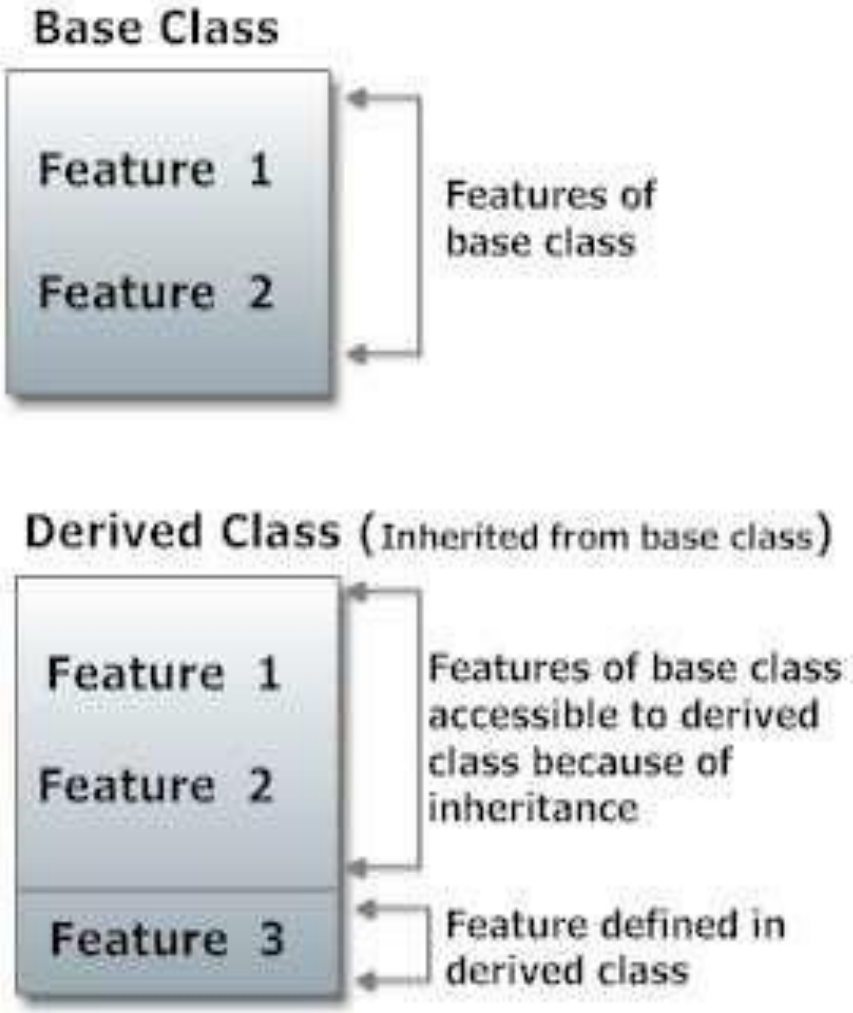
WAP for calculating area of a triangle using public attributes in class

Inheritance

- Inheritance is a **powerful feature** in object oriented programming
- It refers to **defining a new class with little or no modification to an existing class.**
- The new class is called **derived (or child) class** and the one from which it inherits is called the **base (or parent) class.**
- Derived class inherits features from the base class, adding new features to it.
- This results into re-usability of code.

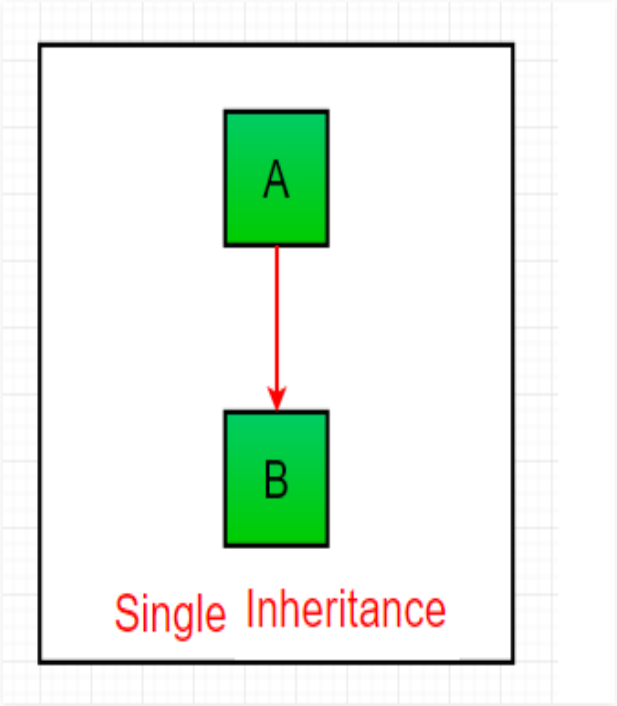
Syntax:

```
class Baseclass(Object):
        body_of_base_class
class DerivedClass(BaseClass):
        body_of_derived_clas
```


Base Class
Feature 1
Feature 2
Features of base class
Derived Class (Inherited from base class)
Feature 1
Feature 2
Features of base class accessible to derived class because of inheritance
Feature 3
Feature defined in derived class

TYPES OF INHERITANCE

Single Inheritance: Single inheritance enables a derived class to inherit properties from a single parent class, thus enabling code reusability and the addition of new features to existing code.



Single inheritance

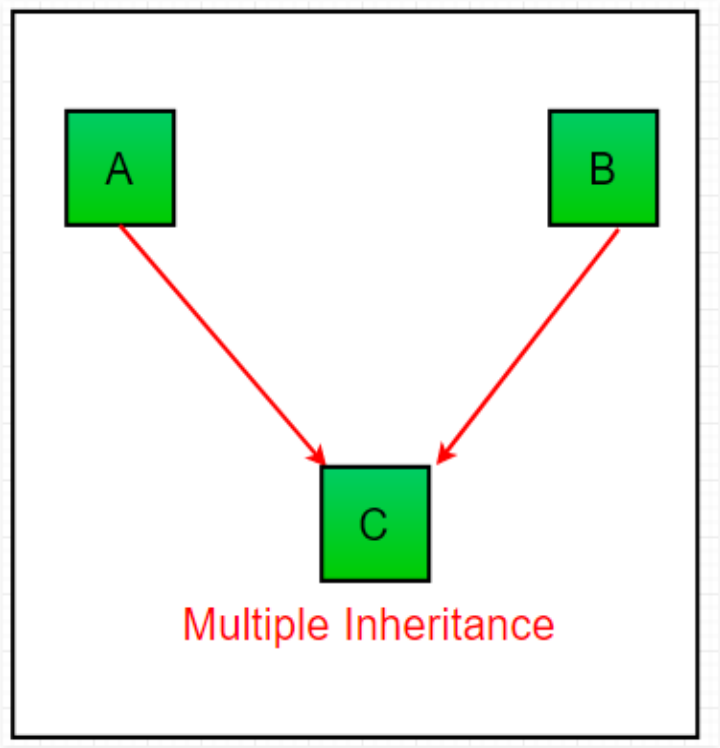
```
# Base class
class Parent:
    def func1(self):
        print("This function is in parent class.")

# Derived class
class Child(Parent):
    def func2(self):
        print("This function is in child class.")

# Driver's code
object = Child()
object.func1()
object.func2()
```

Output:

This function is in parent class.

This function is in child class.

Multiple Inheritance: When a class can be derived from more than one base class this type of inheritance is called multiple inheritance. In multiple inheritance, all the features of the base classes are inherited into the derived class.



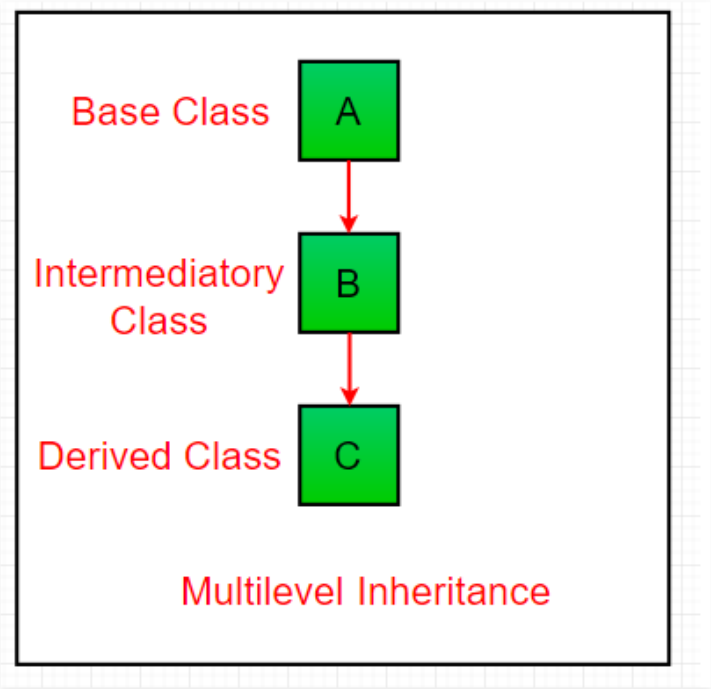
```
# multiple inheritance
# Base class1
class Mother:
    mothername = ""
    def mother(self):
        print(self.mothername)

# Base class2
class Father:
    fathername = ""
    def father(self):
        print(self.fathername)

# Derived class
class Son(Mother, Father):
    def parents(self):
        print("Father :", self.fathername)
        print("Mother :", self.mothername)

# Driver's code
s1 = Son()
s1.fathername = "RAM"
s1.mothername = "SITA"
s1.parents()
```

Output:

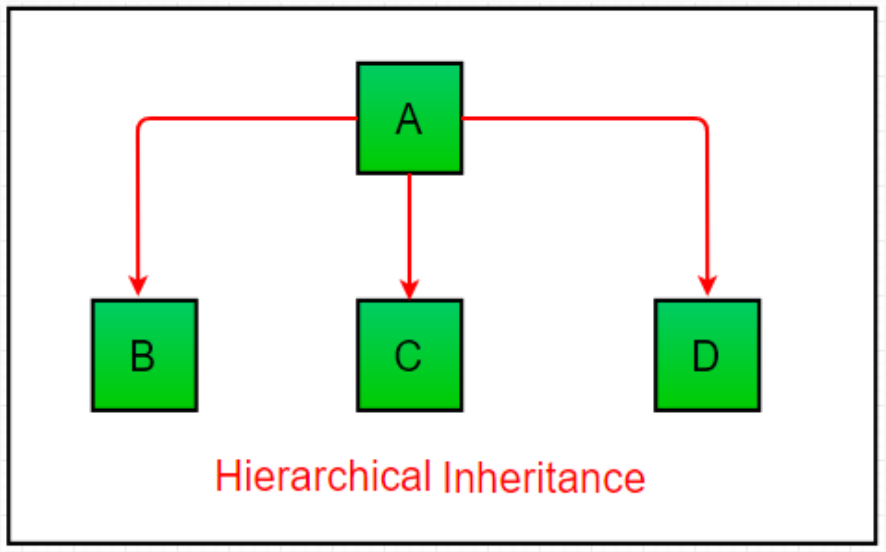
Father : RAM
Mother : SITA

Multilevel Inheritance

In multilevel inheritance, features of the base class and the derived class are further inherited into the new derived class. This is similar to a relationship representing a child and grandfather.



Hierarchical Inheritance: When more than one derived classes are created from a single base this type of inheritance is called hierarchical inheritance. In this program, we have a parent (base) class and two child (derived) classes.



Hybrid Inheritance: Inheritance consisting of multiple types of inheritance is called hybrid inheritance.

Classwork:

Create class disease(d_id,d_name,d_symptoms) and inherit in class medicine(m_id,m_name,m_price,m_quantity) using single inhertance

Create class to store medicinal plant database using hierarchical Inheritance

Polymorphism

- Polymorphism in Latin word which made up of ***'ploy'*** means many and ***'morphs'*** means forms
- From the Greek , Polymorphism means ***many(poly) shapes (morph)***
- Generally speaking, polymorphism means that a method or function is able to cope with different types of input.

In OOP , Polymorphism is the characteristic of being able to assign a different meaning to a particular symbol or operator in different contexts specifically to allow an entity such as a variable, a function or an object to have more than one form.

There are two kinds of Polymorphism

Overloading :

Two or more methods with different signatures

Overriding :

Replacing an inherited method with another having the same signature

```
# method overloading
class Compute:
        # area method
        def area(self, x = None, y = None):
                if(x != None and y != None):
                        return x * y
                elif(x != None):
                        return x * x
                else:
                        return 0
                # object
obj = Compute()
                # zero argument
print("Area Value:", obj.area())
                # one argument
print("Area Value:", obj.area(4))
                # two argument
print("Area Value:", obj.area(3, 5))
```

Note: Method Overloading, a way to create multiple methods with the same name but different arguments direct way, is not possible in Python.

method overriding

```
class Bird:
  def intro(self):
    print("There are many types of birds.")

  def flight(self):
    print("Most of the birds can fly but some cannot.")

class sparrow(Bird):
  def flight(self):
    print("Sparrows can fly.")

class ostrich(Bird):
  def flight(self):
    print("Ostriches cannot fly.")

obj_bird = Bird()
obj_spr = sparrow()
obj_ost = ostrich()

obj_bird.intro()
obj_bird.flight()

obj_spr.intro()
obj_spr.flight()

obj_ost.intro()
obj_ost.flight()
```

Output:

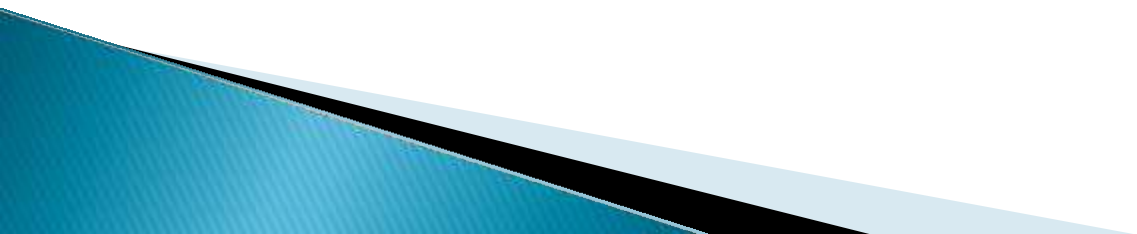
There are many types of birds.
Most of the birds can fly but some cannot.
There are many types of birds.
Sparrows can fly.
There are many types of birds.
Ostriches cannot fly.

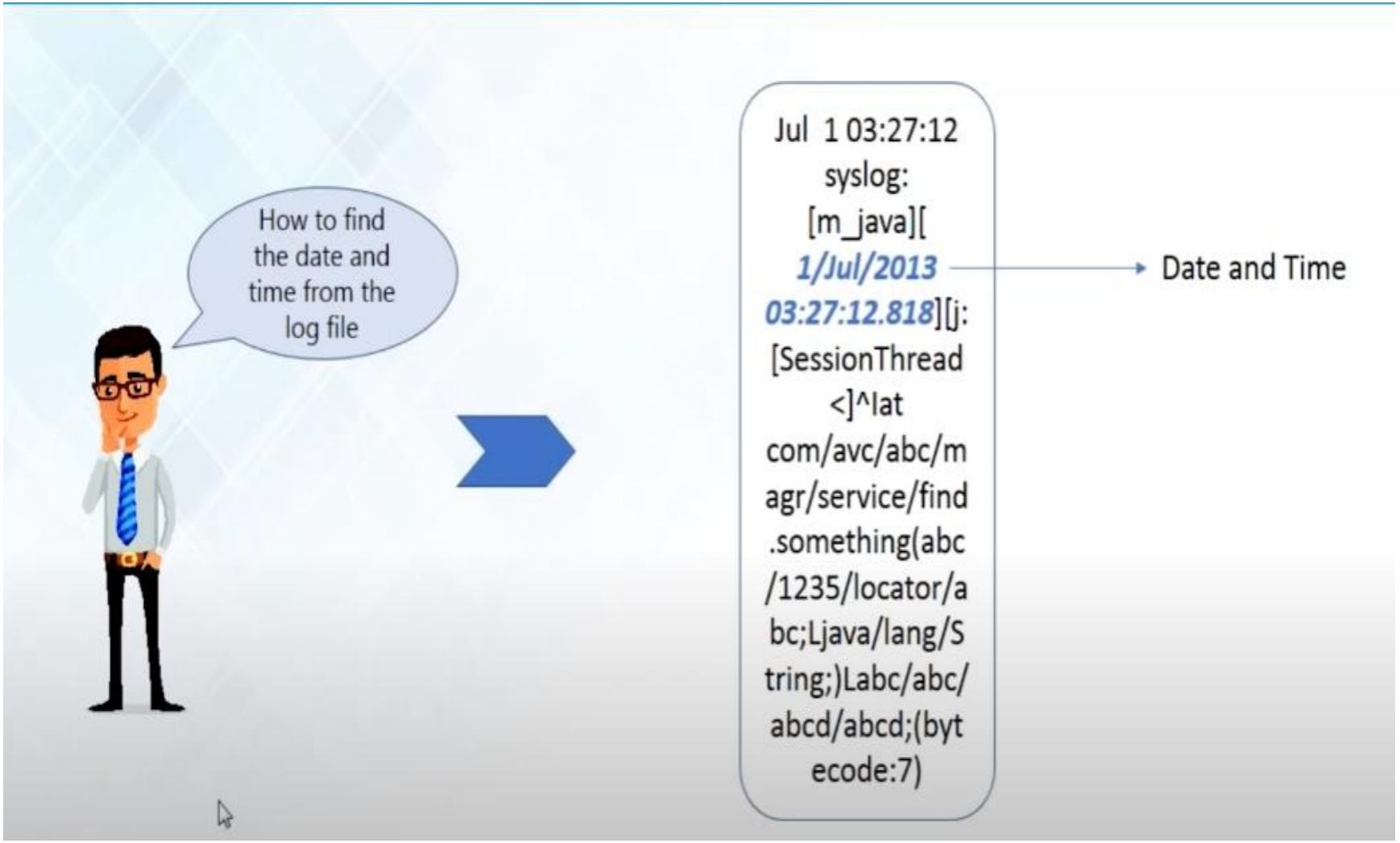
Classwork:

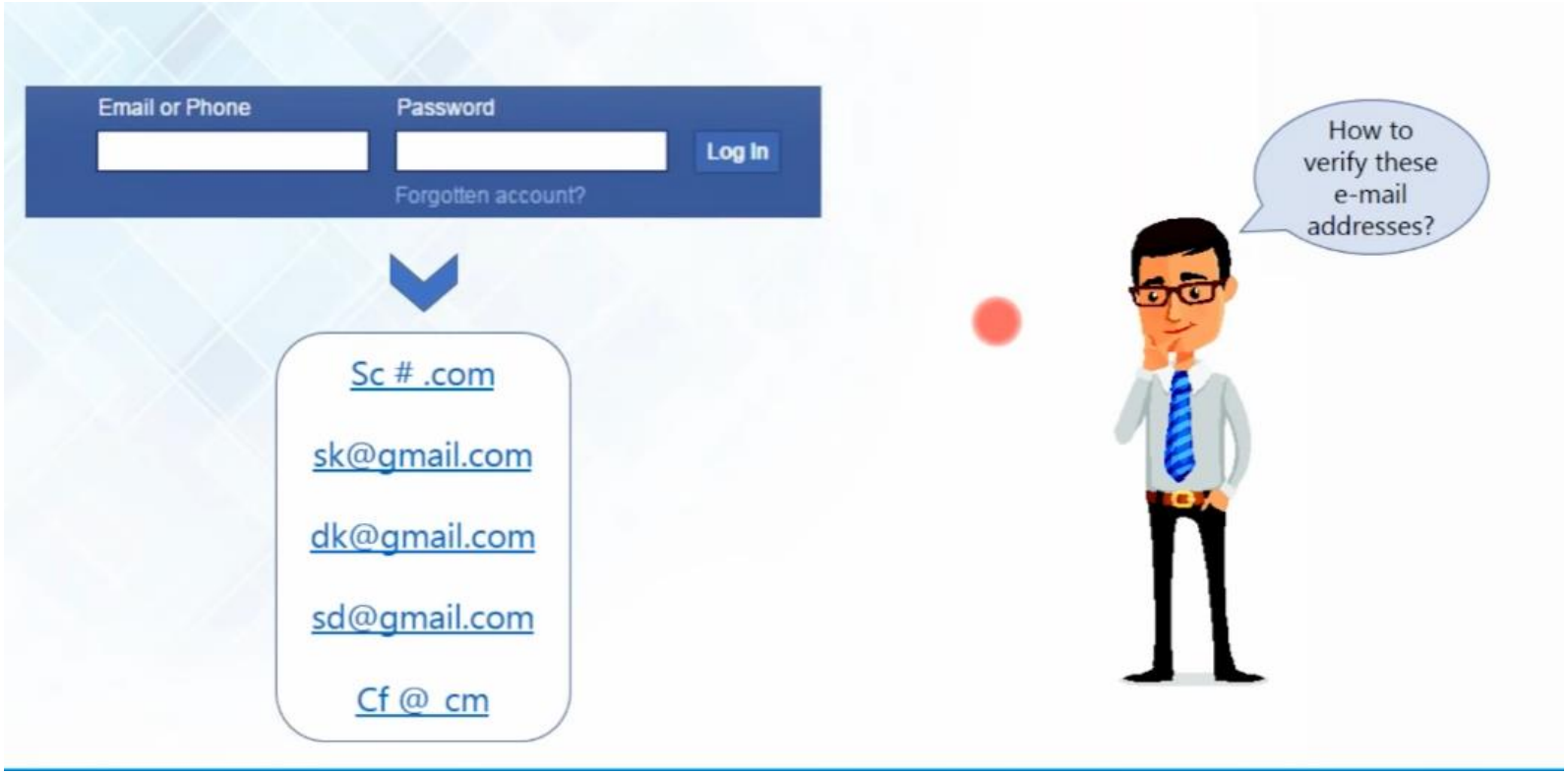
Create class addition for adding two and three numbers using method overloading 12

Python REGULAR EXPRESSIONS

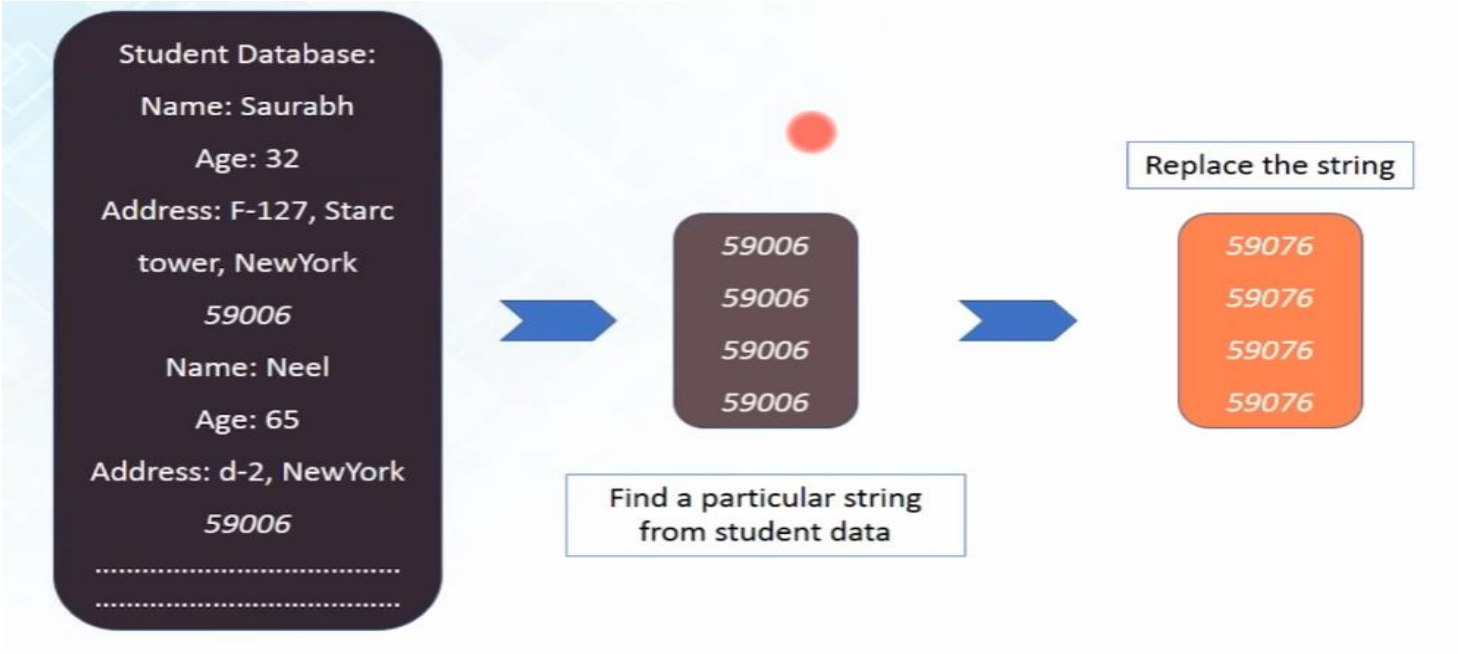
Why we use Regular Expressions?




How to find the date and time from the log file
Jul 1 03:27:12 syslog: [m_java][1/Jul/2013 03:27:12.818][j: [SessionThread <]^Iat com/avc/abc/magr/service/find.something(abc/1235/locator/abc;Ljava/lang/String;)Labc/abc/abcd/abcd;(bytecode:7)
Date and Time


Email or Phone
Password
Log In
Forgotten account?
Sc # .com
sk@gmail.com
dk@gmail.com
sd@gmail.com
Cf @ cm
How to verify these e-mail addresses?


Phone no.:
444-123-2344
10-1029-4562
109-2937-36
100-2939-9390
9292-2929-47
How to verify the phone number and find the country to which it belongs?


Student Database:
Name: Saurabh
Age: 32
Address: F-127, Starc
tower, NewYork
59006
Name: Neel
Age: 65
Address: d-2, NewYork
59006
59006
59006
59006
59006
Find a particular string
from student data
Replace the string
59076
59076
59076
59076

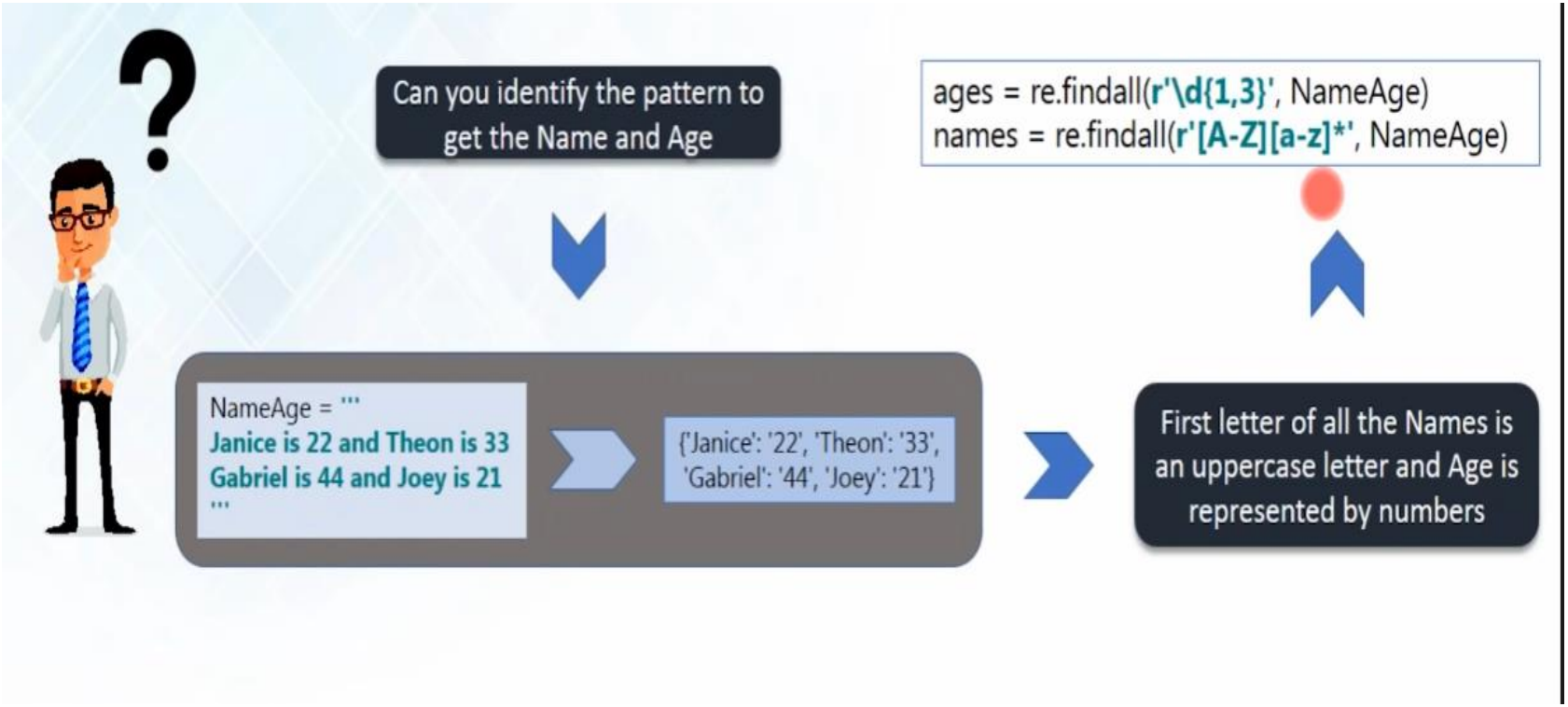
What are Regular Expressions?

A Regular Expression is a special text string for describing a search pattern.

Can you identify the pattern to get the Name and Age

```
NameAge = '''
Janice is 22 and Theon is 33
Gabriel is 44 and Joey is 21
'''
```

{'Janice': '22', 'Theon': '33', 'Gabriel': '44', 'Joey': '21'}


Can you identify the pattern to get the Name and Age
ages = re.findall(r'\d{1,3}', NameAge)
names = re.findall(r'[A-Z][a-z]*', NameAge)
NameAge = '''
Janice is 22 and Theon is 33
Gabriel is 44 and Joey is 21
'''
{'Janice': '22', 'Theon': '33', 'Gabriel': '44', 'Joey': '21'}
First letter of all the Names is an uppercase letter and Age is represented by numbers

Features of Regex

- Hundreds of code could be reduced to a one line elegant regular expression
- Used to construct compilers, interpreters and text editors
- Used to search and match text patterns
- Used to validate text data formats especially input data
- Popular programming languages have Regex capabilities Python, Perl, JavaScript, Ruby ,Tcl, C++,C#,javascript

General uses of regular expressions to:

- Search a string (search and match)
- Replace parts of a string (sub)
- Break string into small pieces (split)
- Finding a string (findall)

Before using the regular expressions in your program, you must import the library using "import re"

General Concepts

- Alternative : |
- Grouping : ()
- Quantification : ?*+{m,n}
- Anchors : ^ $
- Meta- characters : . [][-][^]
- Character classes : \d\D\w\W…..

Alternative:

Eg: “cat|mat” ==“cat” or “mat”

“python|jython” ==“python” or “jython”

Grouping:

Eg: gr(r|a)y==“grey” or “gray”

“ra(mil|n(ny|el))”==“ramil” or “ranny” or “ranel”

Quantification

? == zero or one of the preceding element

Eg: “rani?el”==“raniel” or “ranel”

“colou?r”==“colour” or “color”

* == zero or more of the preceding element

Eg: “fo*ot”==“foot” or “fooot” or “fooooooot”

“94*9” ==“99” or “9449” or “9444449”

+ == one or more of the preceding element

Eg: “too+fan”==“toofan” or “toooooofan”

“36+40”==“3640” or “3666640”

{m,n} == m to n times of the preceding element

Eg: “go{2,3}gle”== “google” or “gooogle”

“6{3} “==“666”

“s{2,}”==“ss” or “sss” or “ssss” ……

Anchors

^ == matches the starting position with in the string

Eg: "^obje"=="object" or "object – oriented"

"^2014"=="2014" or "2014/20/07"

$ == matches the ending position with in the string

Eg: "gram$"=="program" or "kilogram"

"2014$" == "20/07/2014", "2013-2014"

Meta-characters

.(dot)== matches any single character

Eg: “bat.”== “bat” or “bats” or “bata”

“87.1”==“8741” or “8751” or “8761”

[]== matches a single character that is contained with in the brackets

Eg: “[xyz]” == “x” or “y” or “z”

“[aeiou]”==any vowel

“[0123456789]”==any digit

[-] == matches a single character that is contained within the brackets and the specified range.

Eg: “[a-c]” == “a” or “b” or “c”

“[a-zA-Z]” == all letters (lowercase & uppercase)

“[0-9]” == all digits

[^] == matches a single character that is not contained within the brackets.

Eg: “[^aeiou]” == any non-vowel

“[^0-9]” == any non-digit

“[^xyz]” == any character, but not “x”, “y”, or “z”

Character Classes

Character classes specifies a group of characters to match in a string

- \d → Matches a decimal digit [0-9]
- \D → Matches non digits
- \s → Matches any white space character **[\t-tab,\n-newline, \f-form]**
- \S → Matches any non-white space character
- \w → Matches alphanumeric character class ([a-zA-Z0-9_])
- \W → Matches non-alphanumeric character class ([^a-zA-Z0-9_])
- \w+ → Matches one or more words / characters
- \A → Matches beginning of string
- \z → Matches end of string
- \b → Returns a match where the specified characters are at the beginning or at the end of a word
- \B → Returns a match where the specified characters are present, but NOT at the beginning (or at the end) of a word

The search function

Search scans through the input string and tries to match at any location

The search function searches for first occurrence of RE *pattern* within *string* with optional *flags*.

syntax :

re.search(pattern, string, flags=0)

Pattern --This is the regular expression to be matched

String-- This is the string, which would be searched to match the pattern anywhere in the string

Flags-- You can specify different flags using bitwise OR (|). These are modifiers.

Modifier	Description
re.I	Performs case-insensitive matching
re.L	Interprets words according to the current locale. This interpretation affects the alphabetic group (\w and \W,\b,\B).
re.M	Makes $ match the end of a line (not just the end of the string) and makes ^ match the start of any line (not just the start of the string).
re.S	Makes a period (dot) match any character, including a newline.
re.X	Allow comment in Regex

OPTION FLAGS

Example 1:
Search the string to see if it starts with "The" and ends with "Spain":

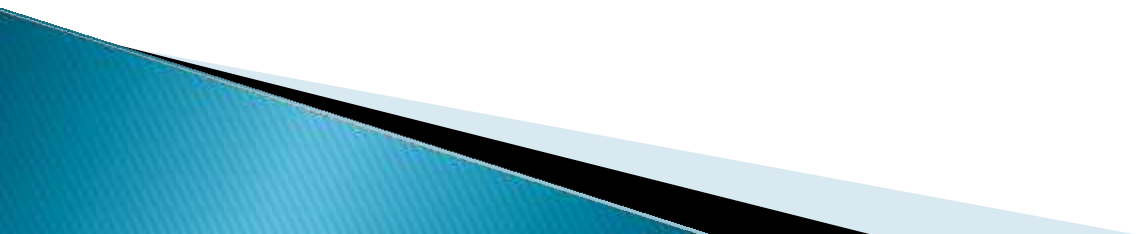
```
import re
txt = "The rain in Spain"
x = re.search("^The.*Spain$", txt)
if x:
  print("YES! We have a match!")
else:
  print("No match")
```

Example 2:

```
import re
# importing Regular Expression built-in module
text = 'This is my First Regulr Expression Program'
patterns = [ 'first', 'that‘, 'program' ]
# In the text input which patterns you have to search

for pattern in patterns:
    if re.search(pattern, text,re.I):
        print 'found a match!'
    else:
        print 'no match'
     # if pattern not found in the text then execute the 'else' condition
```

ClassWork:

1. Search a pattern who as digit 3 to 8 only
2. Search a pattern “ag” from a sequence dna by ignoring case
3. Search a pattern codon TTT or CTT in a sequence



- *match object* for information about the matching string. *match object* instances also have several methods and attributes; the most important ones are

Method	Purpose
group( )	Return the string matched by the RE
start( )	Return the starting position of the match
end ( )	Return the ending position of the match
span ( )	Return a tuple containing the (start, end) positions of the match

Example:

```
import re

txt = "The rain in Spain"
x = re.search("r", txt)

print(“The matched string are :", x.group())
```

Classwork

Search a pattern “ATGACA” from a entered dna sequence by ignoring case

1. Display the searched pattern
2. Return the starting position of the match
3. Return the end position of the match
4. Return a tuple containing the (start, end) positions of the match

Findall:

- Return all non-overlapping matches of *pattern* in *string*, as a list of strings.
- The *string* is scanned left-to-right, and matches are returned in the order found.
- If one or more groups are present in the pattern, return a list of groups; this will be a list of tuples if the pattern has more than one group.
- Empty matches are included in the result unless they touch the beginning of another match.

Syntax:

re.findall (*pattern, string, flags=0*)

The findall example

Example 1:
Print a list of all matches:

```
import re

txt = "The rain in Spain"
x = re.findall("ai", txt)
print(x)
```

Example 2:
Return an empty list if no match was found:

```
import re

txt = "The rain in Spain"
x = re.findall("Portugal", txt)
print(x)
```

Write a regex code for the following string “From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008 @hggg”

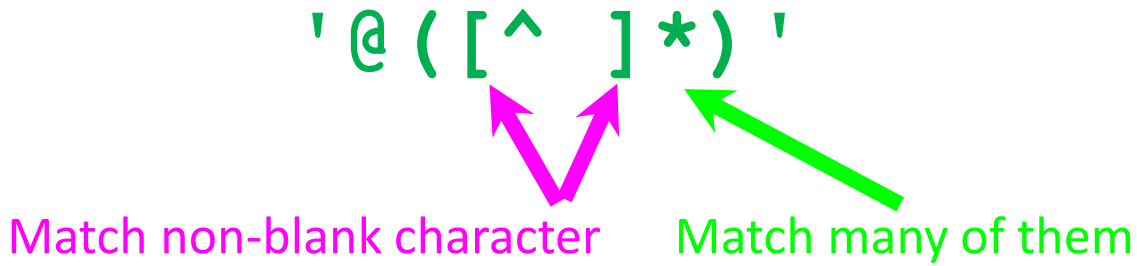
Find wherever I found @ and return the string after that until I find space

Output:
uct.ac.za
hggg

The Regex Version

From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008

```
import re
lin = 'From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008'
y = re.findall('@([^ ]*)',lin)
print (y)
```



Python Raw Strings

- Python Raw Strings are string literals prefixed with a "r" or "R". For example, r"Hello" is a raw string.
- Raw Strings do not treat backslashes ("\") as part of an escape sequence. It will be printed normally as a result.
- This feature can help us pass string literals which cannot be decoded using normal ways, like the sequence "\x".

```
raw_string = r"Hello from\tAskPython"
```

Example: using raw string

```
import re

s = r"Hello\tfrom AskPython\nHi"
print(s)
```

Example: without using raw string

```
import re

s = "Hello\tfrom AskPython\nHi"
print(s)
```

The re.compile() method
re.compile(pattern):

We can combine a regular expression pattern into pattern objects, which can be used for pattern matching. It also helps to search a pattern again without rewriting it.

Example:

```
import re
pattern=re.compile('TP')
result=pattern.findall('TP Tutorialspoint TP')
print result
result2=pattern.findall('TP is most popular tutorials site of India')
print result2
```

Output

```
['TP', 'TP']
['TP']
```

Here we are supposed to verify the phone numbers

Problem Statement:

To verify the phone numbers

Phone Number
444-122-1234
123-122-78999
111-123-23
67-7890-2019

All phone numbers should have:

- 3 starting digits and '-' sign
- 3 middle digits and '-'sign
- 4 digits in the end

Modifying Strings

- Till now we have done simply performed searches against a static string. Regular expressions are also commonly used to modify strings in various ways using the following pattern methods.

Method	Purpose
Split ( )	Split the string into a list, splitting it wherever the RE matches
Sub ( )	Find all substrings where the RE matches, and replace them with a different string
Subn ( )	Does the same thing as sub(), but returns the new string and the number of replacements

Regular expressions are compiled into pattern objects, which have methods for various operations such as searching for pattern matches or performing string substitutions

Split Method:

Syntax:

re.split(string,[maxsplit])

Eg 1:

```
txt = "hello, my name is Peter, I am 26 years old"
x = txt.split(", ")
print(x)
Output:
['hello', 'my name is Peter', 'I am 26 years old']
```

Eg 2:

```
txt = "apple#banana#cherry#orange"
x = txt.split("#", 1)
print(x)
```

Output:

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

Sub Method:

The sub method replaces all occurrences of the RE *pattern* in *string* with *repl*, substituting all occurrences unless *max* provided. This method would return modified string

Syntax:

re.sub(pattern, repl, string, max=0)

Sample Program for Sub Method

```
import re
DOB = "25-01-1991 # This is Date of Birth”
# Delete Python-style comments
Birth = re.sub (r'#.*$', "", DOB)
print "Date of Birth : ", Birth
# Remove anything other than digits
Birth1 = re.sub (r'\D', "", Birth)
print "Before substituting DOB : ", Birth1
# Substituting the '-' with '.(dot)'
New=re.sub (r'\W',".",Birth)
print "After substituting DOB: ", New
```

Write a python script to substitute empty space in a sequence wherever it find digit

Text are :atgg1ttag6gg

Finditer

- Return an *iterator* yielding MatchObject instances over all non-overlapping matches for the RE *pattern* in *string*.
- The *string* is scanned left-to-right, and matches are returned in the order found.
- Empty matches are included in the result unless they touch the beginning of another match.

Syntax:

re.finditer(*pattern, string, flags=0*)

findall and finditer program

```
import re
string="Python java c++ perl shell ruby tcl  c#"
print (re.findall(r"\bc[\W+]*",string,re.M|re.I))
print (re.findall(r"\bp[\w]*",string,re.M|re.I))
print (re.findall(r"\bs[\w]*",string,re.M|re.I))
print (re.sub(r'\W+',"",string))
it = re.finditer(r"\bc[(\W\s)]*", string)
for match in it:
    print ("'{gh}' was found between the
indices{st}".format(gh=match.group(),st=match.span()))
```

OUTPUT:

['c++ ', 'c#']

['Python', 'perl']

['shell']

Pythonjavacperlshellrubytclc

'c++ ' was found between the indices(12, 16)
'c#' was found between the indices(37, 39)

C/W:

Q.1) Write a python script to store a string in a variable as “Jessa is a Python developer. She also gives Python programming training”. Perform the following:

1. find all words that start ‘p’and ends with a ‘g’
2. find all words that start with a specific substring ‘Py’
3. find all words that contains with a specific substring ‘ing’ anywhere

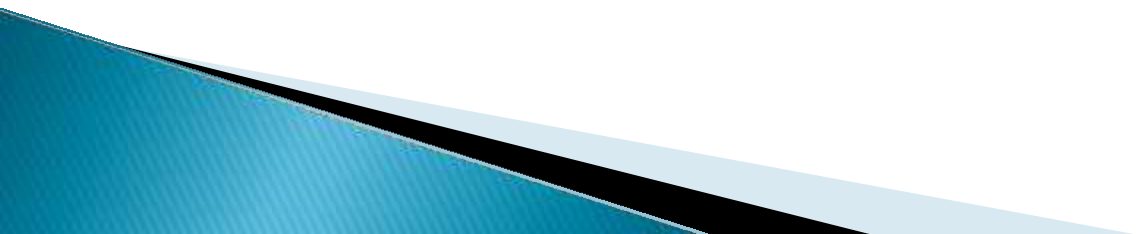
H/W

Q.2)

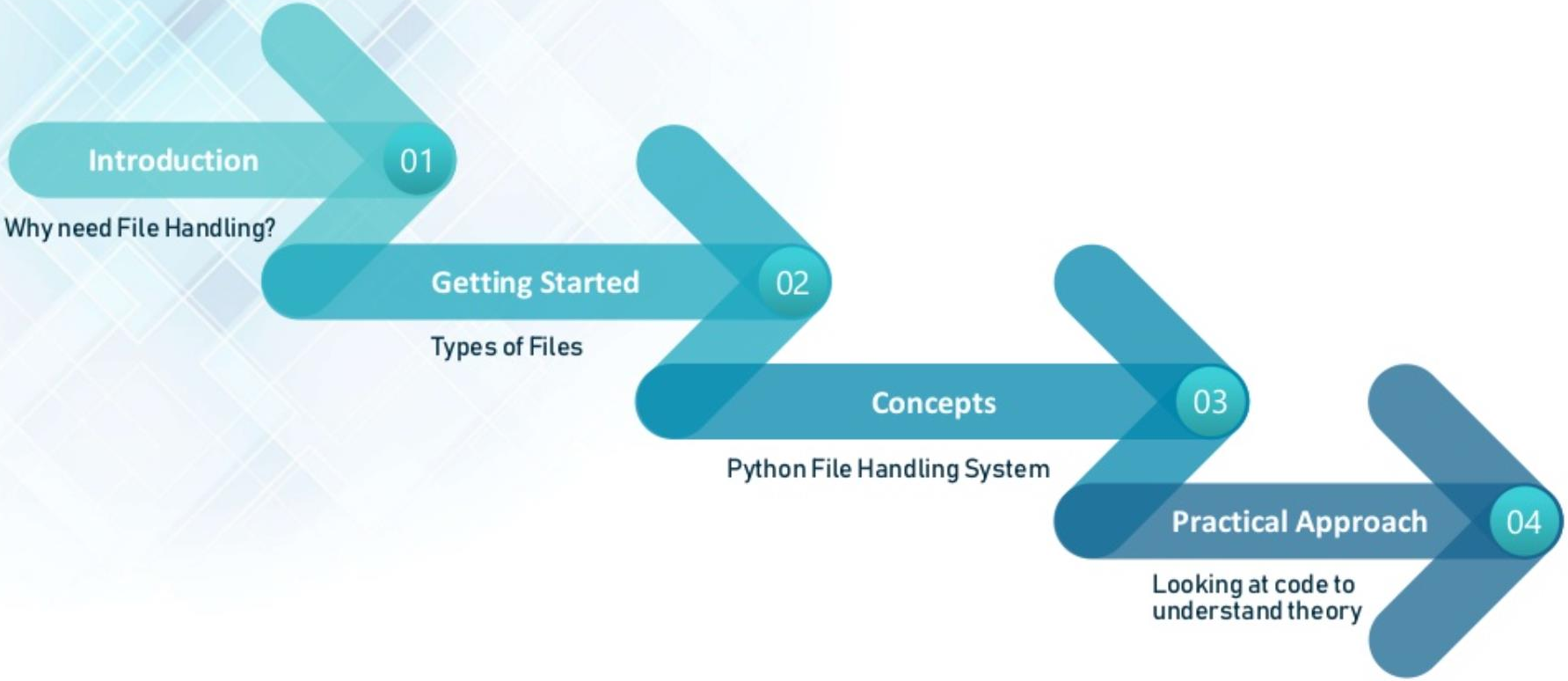
Conditions for a valid password are:

1.Should have at least one number.
2.Should have at least one uppercase and one lowercase character.
3.Should have at least one special symbol.
4.Should be between 6 to 20 characters long.

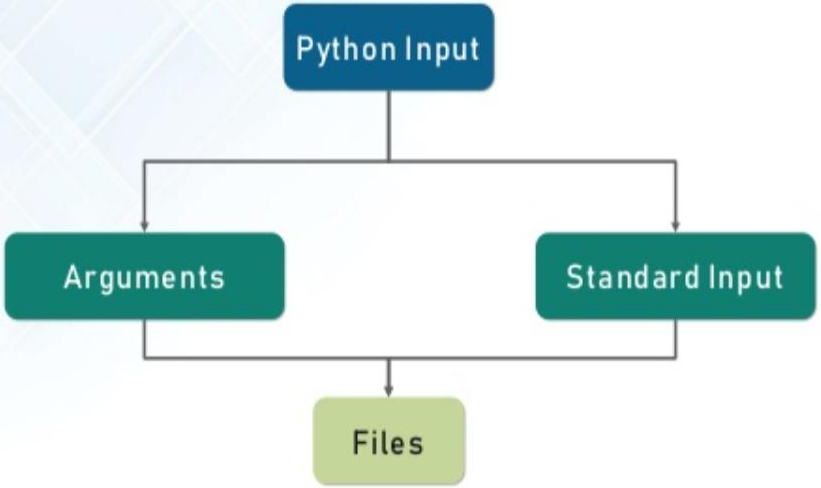
File Handling In Python



Agenda


Introduction
01
Why need File Handling?
Getting Started
02
Types of Files
Concepts
03
Python File Handling System
Practical Approach
04
Looking at code to understand theory

Why Need File Handling?



Types Of Files

What you may know as a file is slightly different in Python

Image

Text

Executable

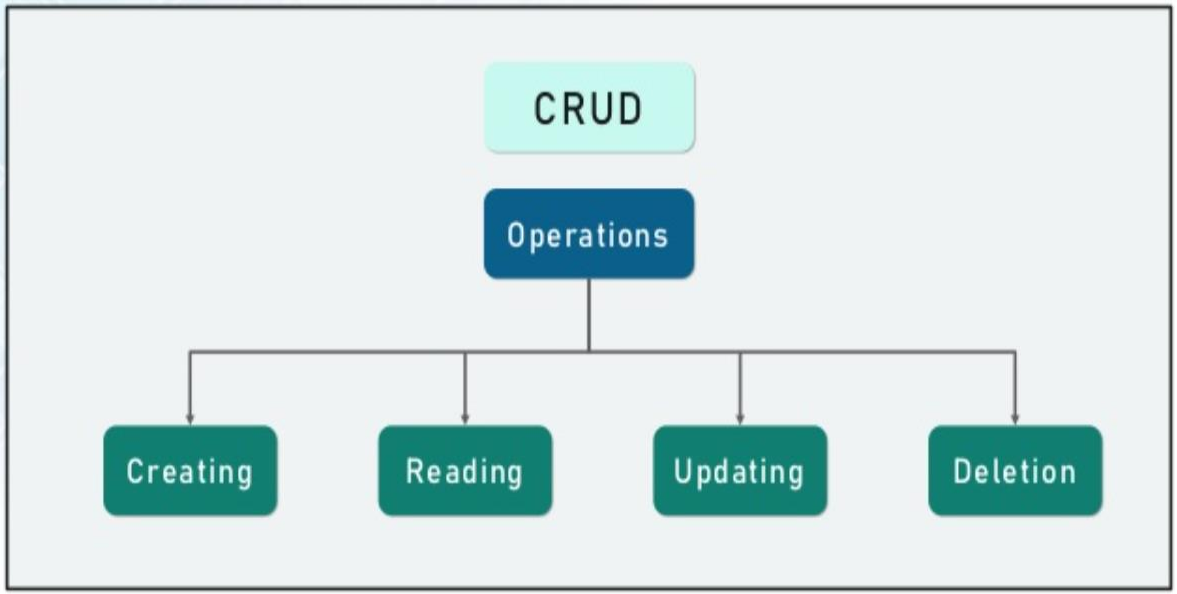
Audio

Binary

Text

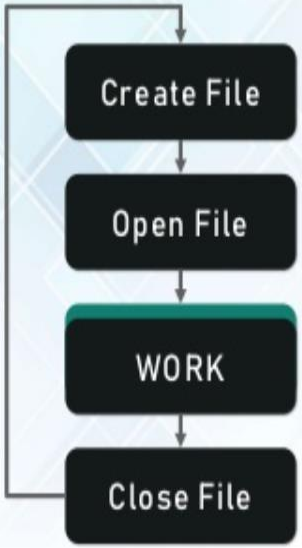
What Is File Handling?

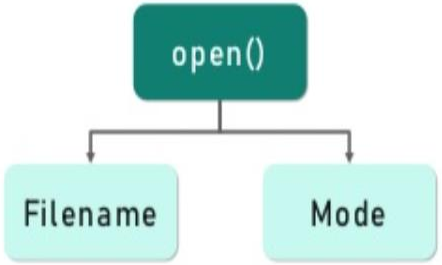
File handling is an important part of any web application



Python File Handling System

The key function for working with files in Python is the open() function





Syntax ⇨ open(filename, mode)

Python File Handling System

The key function for working with files in Python is the open() function

Syntax ⇨ open(filename, mode)

Any name that you want

Different modes for opening a file

"r" - Read - Default value. Opens a file for reading, error if the file does not exist

"a" - Append - Opens a file for appending, creates the file if it does not exist

"w" - Write - Opens a file for writing, creates the file if it does not exist

"x" - Create - Creates the specified file, returns an error if the file exists

In addition you can specify if the file should be handled as binary or text mode

"t" - Text - Default value. Text mode

"b" - Binary - Binary mode (e.g. images)

Python File Handling System

Example Code

Example ⇨ f = open("demofile.txt")

=

Example ⇨ f = open("demofile.txt", "r")

Note: Make sure file exists or else error!

File Handling In Python

File Operations for Reading

Reading Text File In Python

file.read()

Lots of ways to read a text file in Python

All characters

Some characters

Example ⇨

```
> file = open("testfile.text", "r")
> print file.read()
```

Reading Text File In Python

file.read()

Example ⇨

```
> file = open("testfile.text", "r")
> print file.read(5)
```

→ This 5 indicates what?

Example ⇨

```
> file = open("testfile.text", "r")
> print file.read()
```

Reading Text File In Python

file.read()

Example

```
> file = open("testfile.text", "r")
> print file.readline():
```

→ Line by line output

Example

```
> file = open("testfile.text", "r")
> print file.readline(3):
```

→ Read third line only

Looping Over A File Object

Fast and efficient!

Example

```
> file = open("testfile.text", "r")
> for line in file:
>     print file.readline():
```

Looping over the object

Reading from files


File Handling In Python
Python File Write Method

File Write Method

Writing to an existing file

To write to an existing file, you must add a parameter to the open() function:

"a" - Append - will append to the end of the file

"w" - Write - will overwrite any existing content

Example ⇨

```
> f = open("demofile.txt", "a")
> f.write(" We love Edureka!")
```

Example ⇨

```
> f = open("demofile.txt", "w")
> f.write(" We love Edureka!")
```

Note: the "w" method will overwrite the entire file.

File Write Method

Example

```
> file = open("testfile.txt", "w")

> file.write("This is a test")
> file.write("To add more lines.")

> file.close()
```

I'm writing files!

File Handling In Python

Creating a New File

Creating A New File

open() method again

To create a new file in Python, use the open() method, with one of the following parameters:

"x" - Create - will create a file, returns an error if the file exist

"a" - Append - will create a file if the specified file does not exist

"w" - Write - will create a file if the specified file does not exist

```
file = open("testfile.txt", "x")
file = open("testfile.txt", "w")
```

File Handling In Python

Deletion Operations

Deleting A File

os.remove() function

To delete a file, you must import the OS module, and run its os.remove() function:

Example ⇨

```
> import os
> os.remove("demofile.txt")
```

Check if file exists

```
> import os
> if os.path.exists("demofile.txt"):
>    os.remove("demofile.txt")
> else:
>    print("The file does not exist")
```

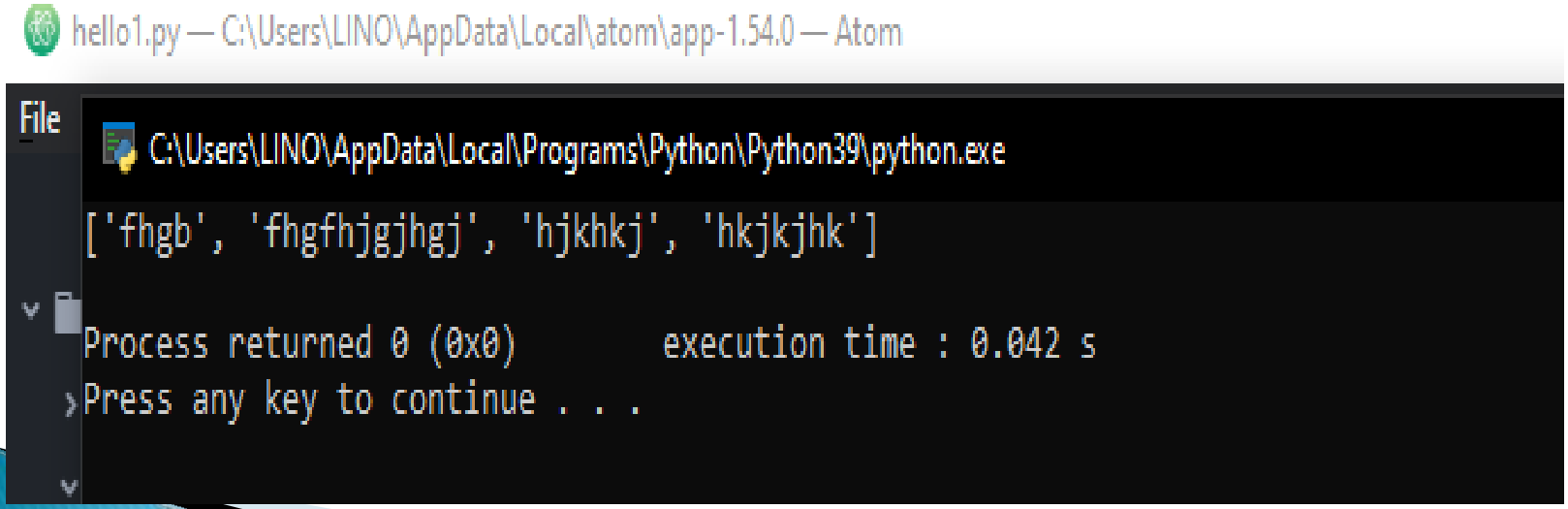
Deleting a folder?

Example ⇨

```
> import os
> os.rmdir("myfolder")
```

Write a regex code for matching the non digits and display the list using file handling.

```
import re
textfile=open("C:\\Users\\LINO\Desktop\\file\\abc.txt","r")
matches = []
reg = re.compile("[^\d]+")
for line in textfile:
    matches += reg.findall(line)
textfile.close()
print(matches)
```




Questions

Classwork :

1. *Create a text file "MyFile.txt" in python and ask the user to write separate 3 lines with three input statements from the user.*
2. *Write a program to count a total number of lines. Find the total occurrences of a specific word from a text file entered by user.*
3. *WAPS to validate the sequence as dna in a seqfile.fasta*

THANK YOU

UNIT 3: Biopython

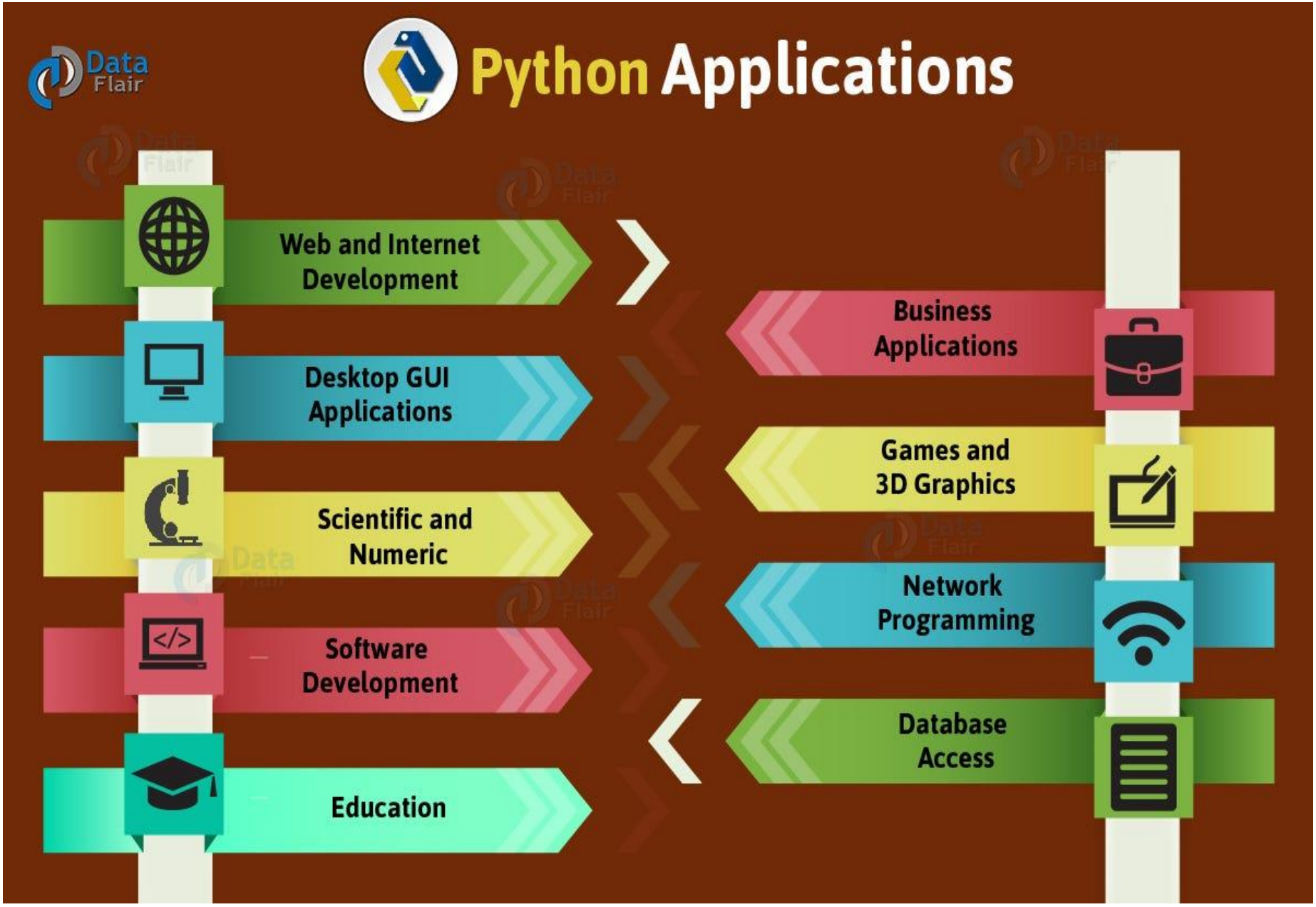
- Biopython is an integrated collection of modules for “biological computation” including tools for working with
  - DNA/protein sequences,
  - sequence alignments,
  - population genetics, and
  - molecular structures
- It also provides interfaces to common biological databases (i.e. GenBank) and to some common locally installed software (i.e. BLAST).
- Its an open-source

Features

- Interpreted, interactive and object oriented.
- Supports FASTA, PDB, GenBank, Blast, SCOP, PubMed/Medline, ExPASy-related formats.
- Tools to manage protein structures.
- BioSQL
- Access to online services and database, including NCBI services (Blast, Entrez, PubMed) and ExPASY services (SwissProt, Prosite).
- Access to local services, including Blast, Clustalw, EMBOSS.

What can you do with Biopython?

- Read, write & manipulate sequences
- Restriction enzymes
- BLAST (local and online)
- Web databases (e.g. NCBI's EUtils)
- Call command line tools (e.g. clustalw)
- Clustering (Bio.Cluster)
- Phylogenetics (Bio.Nexus)
- Protein Structures (Bio.PDB)


Data Flair
Python Applications
Web and Internet Development
Desktop GUI Applications
Scientific and Numeric
Software Development
Education
Business Applications
Games and 3D Graphics
Network Programming
Database Access

Biopython history

- 1999 : Started by Jeff Chang & Andrew Dalke
- 2000 : Biopython 0.90, first release
- 2001 : Biopython 1.00, "semi-complete"
- 2002 : Biopython 1.10, "semi-stable"
- 2003 : Biopython 1.20, 1.21, 1.22 and 1.23
- 2004 : Biopython 1.24 and 1.30
- 2005 : Biopython 1.40 and 1.41
- 2006 : Biopython 1.42
- 2007 : Biopython 1.43

Note: Latest Release

28 June 2024: biopython 1.84

How to Install Biopython

First download python 3.3 :
https://www.python.org/downloads/release/python-330/

Than download biopython:
https://biopython.org/wiki/Download

Install python 3.3 first than install biopython→by clicking on next next button

or

On cmd
>>pip3.11 install biopython

Object oriented programming

- Biopython is **object-oriented**
- OOP is a way of **organizing data and methods that work on them in a coherent package**
- OOP helps **structure and organize the code**

Biopython - Sequence

- A sequence is series of letters used to **represent an organism's protein, DNA or RNA.**
- It is represented by **Seq class**.
- Seq class is defined in **Bio.Seq module**.

EXAMPLE−

```
>>> from Bio.Seq import Seq
>>> seq = Seq("AGCT")
>>> seq
Seq('AGCT')
>>> print(seq)
AGCT
```

The Alphabet object

- alphabet – used to represent the type of sequence. e.g. DNA sequence, RNA sequence, etc.
- The alphabets are actually defined objects such as `IUPAC.unambiguous_dna` or `more`
  - Which are defined in the Bio.Alphabet module
  - A Seq object with a DNA alphabet has some different methods than one with an Amino Acid alphabet

International Union of Pure and Applied Chemistry

This command creates the Seq object

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> my_seq = Seq('AGTACACTGGT', IUPAC.unambiguous_dna)
>>> my_seq
Seq('AGTACACTGGT', IUPAC.unambiguous_dna())
>>> print(my_seq)
AGTACACTGGT
```

Basic Operations

We can perform python string operations like slicing, counting, concatenation, find, split and strip in sequences.

Use the below codes to get various outputs.

1. To get the first value in sequence.

```
>>> seq_string = Seq("AGCTAGCT")
>>> seq_string[0]
'A'
```

2. To print the first two values.

```
>>> seq_string[0:2]
Seq('AG')
```

3. To print all the values.

```
>>> seq_string[ : ]
Seq('AGCTAGCT')
```

4. To perform length and count operations.

```
>>> len(seq_string)
8
>>> seq_string.count('A')
2
```

5. To add or concatenate two sequences.

```
>>> from Bio.Alphabet import IUPAC
>>> seq1 = Seq("AGCT", IUPAC.unambiguous_dna)
>>> seq2 = Seq("TCGA", IUPAC.unambiguous_dna)
>>> seq1+seq2
Seq('AGCTTCGA', DNAAlphabet())
```

Turn a Seq object into a string

- Sometimes you will need to work with just the sequence string in a Seq object using a tool that is not aware of the Seq object methods
- Turn a Seq object into a string with str()

```
>>> my_seq
Seq('GATCGATGGGCCTATATAGGATCGAAAATCGC',
IUPACUnambiguousDNA())
>>> seq_string=str(my_seq)
>>> seq_string
'GATCGATGGGCCTATATAGGATCGAAAATCGC'
```

Question:

1. Write a Biopython script to create a dna sequence and calculate a GC content and GC%

Seq Objects have special methods

Seq objects with a DNA alphabet have the reverse_complement() method:

```
>>> my_seq = Seq('TTTAAAATGCGGG')
>>> print(my_seq.reverse_complement())
CCCGCATTTTAAA
```

The Bio.SeqUtils module has some useful methods, such as GC() to calculate % of G+C bases in a DNA sequence.

```
>>> from  Bio.SeqUtils  import  GC
>>> GC(my_seq)
46.875
```

Convert dna to RNA sequence

```
>>>dna_seq('TTTAAAATGCGGG')
>>>rna_seq = transcribe(dna_seq)
```

- DNA Seq objects can .translate() to protein
- Here, the stop codons are indicated with an asterisk ’*’.

```
>>>coding_dna=Seq("ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG")
>>> coding_dna.translate()
Seq('MAIVMGR*KGAR*', HasStopCodon(IUPACProtein(), '*'))
```

Protein Alphabet

- You could re-define my_seq as a protein by changing the alphabet, which will totally change the methods that will work on it.
    - ('G','A','T','C' are valid protein letters)

```
>>> from Bio.SeqUtils import molecular_weight
>>> my_seq
Seq('AGTACACTGGT', IUPACUnambiguousDNA())
>>> print(molecular_weight(my_seq))
3436.1957

>>> my_seq.alphabet = IUPAC.protein
>>> my_seq
Seq('AGTACACTGGT', IUPACProtein())
>>> print(molecular_weight(my_seq))
912.0004
```

Classwork

1. Write a biopython script to store dna sequence using alphabet class.
Perform following operation:

- To get the second value in sequence
- To print the first two values
- To perform length and count number of ‘g’
- Create second sequence and to add two sequences and display
- Turn a Seq object into a string

2. Write a biopython script to store dna sequence.
Perform following operation:

a. Find reverse complement
b. Calculate GC percentage in dna sequence
c. Convert dna to rna
d. Convert dna to protein

SeqRecord Object

- The SeqRecord object is like a database record (such as GenBank). It is a complex object that contains a **Seq** object, and also annotation fields, known as “attributes”.

  .seq
  .id
  .name
  .description
  .letter_annotations
  .annotations
  .features
  .dbxrefs

- You can think of attributes as slots with names inside the SeqRecord object. Each one may contain data (usually a string) or be empty.

SeqRecord Example

```
>>> from Bio.Seq import Seq
>>> from Bio.SeqRecord import SeqRecord
>>> test_seq = Seq('GATC')
>>> test_record = SeqRecord(test_seq, id='xyz')
>>> test_record.description= 'This is only a test'
>>> print(test_record)
ID: xyz
Name: <unknown name>
Description: This is only a test
Number of features: 0
Seq('GATC', Alphabet())
>>> print(test_record.seq)
GATC
```

- Specify fields in the SeqRecord object with a **.**(dot) syntax

Classwork

1. Write a biopython script to store dna sequence as using alphabet class.
Perform following operation:

- To get the second value in sequence
- To print the first two values
- To perform length and count number of ‘g’
- Create second sequence and to add two sequences and display
- Turn a Seq object into a string

2. Write a biopython script to store dna sequence using sequence record.
Perform following operation:

a. Find reverse complement
b. Calculate GC percentage in dna sequence
c. Convert dna to rna
d. Convert dna to protein

SeqIO and FASTA files

- **SeqIO** is the all-purpose file read/write tool for SeqRecords
  - SeqIO can read many file types: http://biopython.org/wiki/SeqIO
- SeqIO has .read() and .write() methods
  - (do not need to "open" file first)
- It can read a text file in FASTA format
- In Biopython, fasta is a type of SeqRecord with specific fields
  - Lets assume you have already downloaded a FASTA file from GenBank, such as: NC_005816.fna, and saved it as a text file in your current directory

```
>>> from Bio import SeqIO
>>> gene = SeqIO.read("NC_005816.fna", "fasta")
>>> gene.id
'gi|45478711|ref|NC_005816.1|'
>>> gene.seq
Seq('TGTAACGAACGGTGCAATAGTGATCCACACCCAACGCCTGAAATCAGATCCAGGCTG',
SingleLetterAlphabet())
>>> len(gene.seq)
9609
```

Multiple FASTA Records in one file

- The FASTA format can store many sequences in one text file
- SeqIO.parse() reads the records one by one
- This code creates a list of SeqRecord objects:

```
>>> from Bio import SeqIO
>>> handle = open("example.fasta", "r")
                                    # “handle” is a pointer to the file
>>> seq_list = list(SeqIO.parse(handle, "fasta"))
>>> handle.close()
>>> print(seq_list[0].seq)          #shows the first sequence in the list
```

SeqIO for FASTQ

- FASTQ is a format for Next Generation DNA sequence data (FASTA + Quality)
- SeqIO can read (and write) FASTQ format files

```
from Bio import SeqIO
count = 0
for rec in SeqIO.parse("SRR020192.fastq", "fastq"):
        count += 1
print(count)
```

Direct Access to GenBank

- BioPython has modules that can **directly access databases over the Internet**
- The Entrez module uses the **NCBI Efetch service**
- Efetch works on many NCBI databases including **protein and PubMed literature citations**
- The '**gb' data type** contains much more annotation information, but rettype='fasta' also works

```
>>> from Bio import Entrez
>>>Entrez.email = "stu@nyu.edu"
                    # NCBI requires your identity
>>> handle = Entrez.efetch(db="nucleotide", id="186972394",
                    rettype="gb", retmode="text")
>>> record = SeqIO.read(handle, "genbank")
```

```
>>> print(record)
ID: EU490707.1
Name: EU490707
Description: Selenipedium aequinoctiale maturase K (matK) gene, partial cds;
chloroplast.
Number of features: 3
/sequence_version=1
/source=chloroplast Selenipedium aequinoctiale
/taxonomy=['Eukaryota', 'Viridiplantae', 'Streptophyta', 'Embryophyta', 'Tracheophyta',
'Spermatophyta', 'Magnoliophyta', 'Liliopsida', 'Asparagales', 'Orchidaceae',
'Cypripedioideae', 'Selenipedium']
/keywords=['']
/references=[Reference(title='Phylogenetic utility of ycf1 in orchids: a plastid gene
more variable than matK', ...), Reference(title='Direct Submission', ...)]
/accessions=['EU490707']
/data_file_division=PLN
/date=15-JAN-2009
/organism=Selenipedium aequinoctiale
/gi=186972394
Seq('ATTTTTTACGAACCTGTGGAAATTTTTGGTTATGACAATAAATCTAGTTTAGTA...GAA',
IUPACAmbiguousDNA())
```

These are sub-fields of the .annotations field

BLAST

Dealing with BLAST can be split up into two steps, both of which can be done from within Biopython.

- Firstly, running BLAST for your query sequence(s), and getting some output.
- Secondly, parsing the BLAST output in Python for further analysis.

BLAST

- BioPython has several methods to work with the popular NCBI BLAST software
- NCBIWWW.qblast() sends queries directly to the NCBI BLAST server. The query can be a Seq object, FASTA file, or a GenBank ID.

We use the function qblast() in the Bio.Blast.NCBIWWW module to call the online version of BLAST. This has three non-optional arguments:

- The first argument is the blast program to use for the search, as a lower case string. The options and descriptions of the programs are available at https://blast.ncbi.nlm.nih.gov/Blast.cgi. Currently qblast only works with blastn, blastp, blastx, tblast and tblastx.
- The second argument specifies the databases to search against. Again, the options for this are available on the NCBI Guide to BLAST ftp://ftp.ncbi.nlm.nih.gov/pub/factsheets/HowTo_BLASTGuide.pdf.
- The third argument is a string containing your query sequence. This can either be the sequence itself, the sequence in fasta format, or an identifier like a GI number.

BLAST

```
>>> from Bio.Blast import NCBIWWW
>>> query = SeqIO.read("test.fasta", format="fasta")
 >>> result_handle = NCBIWWW.qblast("blastn", "nt",
query.seq)
>>> blast_file = open("my_blast.xml", "w")
                              #create an xml output file
>>> blast_file.write(result_handle.read())
>>> blast_file.close()           # tidy up
>>> result_handle.close()
```

Parse BLAST Results

- It is often useful to obtain a BLAST result directly (local BLAST server or via Web browser) and then parse the result file with Python.
- Save the BLAST result in XML format
  - NCBIXML.read() for a file with a single BLAST result (single query)
  - NCBIXML.parse() for a file with multiple BLAST results (multiple queries)

```
>>> from Bio.Blast import NCBIXML
>>> handle = open("my_blast.xml")
>>> blast_record = NCBIXML.read(handle)
>>> for hit in blast_record.descriptions:
        print hit.title
        print hit.e
```

Biopython - Motif Objects

A sequence motif is a nucleotide or amino-acid sequence pattern. Sequence motifs are formed by three-dimensional arrangement of amino acids which may not be adjacent. Biopython provides a separate module, Bio.motifs to access the functionalities of sequence motif as specified below –

Syntax:

from Bio import motifs

Creating Simple DNA Motif

- Let us create a simple DNA motif sequence using the below command −

```
>>> from Bio import motifs
>>> from Bio.Seq import Seq
>>> DNA_motif = [ Seq("AGCT"), Seq("TCGA"), Seq("AACT"),]
seq = motifs.create(DNA_motif)
print(seq)
```

Output:

```
AGCT  TCGA  AACT
```

To count the sequence values, use the below command −

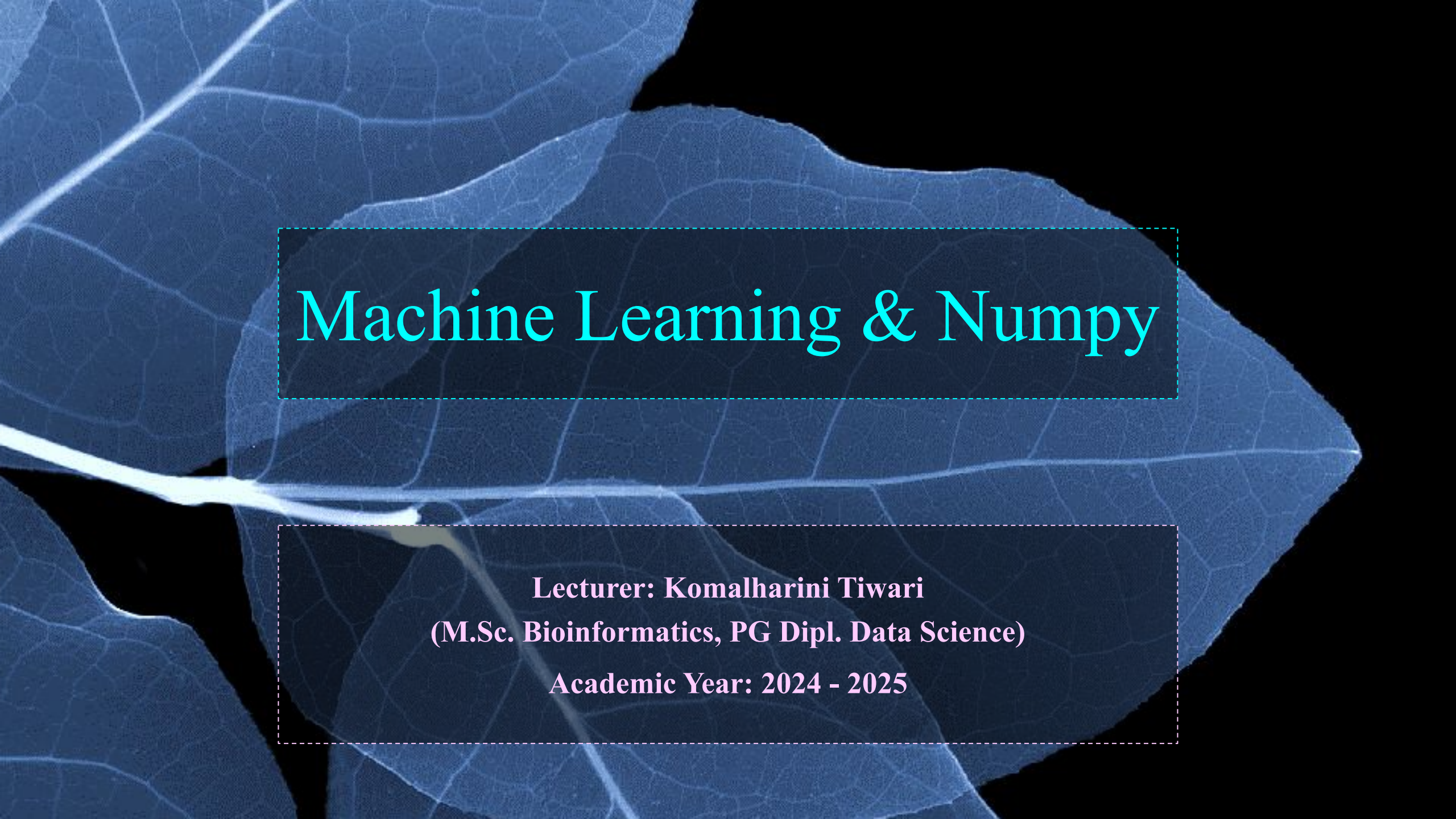
```
>>> print(seq.counts)
        0       1      2       3
A:    2.00    1.00   0.00    1.00
C:    0.00    1.00   2.00    0.00
G:    0.00    1.00   1.00    0.00
T:    1.00    0.00   0.00    2.00

>>> seq.counts["A", :]
(2, 1, 0, 1)
```

C/W:

Write a biopython script to read fasta file and replace the codons ‘ag’ with ‘tc’ then count its nucleotide each.

Learning Objectives

- Biopython is a toolkit
- Seq objects and their methods
- SeqRecord objects have data fields
- SeqIO to read and write sequence objects
- Direct access to GenBank with Entrez.efetch
- Working with BLAST results



Machine Learning & Numpy

Lecturer: Komalharini Tiwari

(M.Sc. Bioinformatics, PG Dipl. Data Science)

Academic Year: 2024 - 2025

Introduction to ML:

- Machine learning is a field of study in artificial intelligence (AI) that focuses on developing algorithms that can learn from data and improve over time.
- It allows computers to learn and make decisions without being explicitly programmed.
- Think of it as teaching a computer to recognize patterns in data so it can make predictions or decisions.
- For example, ML is used in email filtering (to identify spam), recommendation systems (like those on Netflix or Amazon), and even self-driving cars.
- Imagine teaching a child to recognize animals. Initially, you show them pictures of cats and dogs, and over time, the child learns to distinguish between the two. Similarly, in ML, you provide the computer with data, and it learns to identify patterns and make decisions based on that data.
- Ethical considerations:
  - Machine learning raises ethical considerations about the decisions made by advanced models, including transparency, explainability, and their effect on employment and society.
- Applications:
  - Machine learning is used in many industries, including healthcare, finance, retail, travel, and social media. For example, machine learning can help diagnose cancer from X-ray images, identify investment opportunities, and personalize news feeds.

ML in Bioinformatics:

- In bioinformatics, ML is crucial for tasks like protein structure prediction, drug discovery, and disease diagnosis.
- Imagine you're trying to identify a protein's function based on its amino acid sequence. Instead of manually comparing the sequence with known proteins, ML algorithms can be trained to recognize patterns in the sequence and predict its function, speeding up the research process.
- ML in bioinformatics involves training algorithms on large datasets of biological data, such as protein sequences, chemical compounds, or patient records.
- The trained models can then be used to predict outcomes like protein-protein interactions, drug efficacy, or disease risk..

How it works:

- Machine learning algorithms analyze large amounts of data, identify patterns, and make predictions based on that analysis. The more data the algorithm is exposed to, the better it performs.
- ML involves feeding a computer system with large amounts of data and using algorithms to analyze and learn from this data. The computer then uses this learned information to make predictions or decisions on new data.
- Basic Steps:
  - Data Collection: Gather data that you want the machine to learn from.
  - Data Preprocessing: Clean and prepare the data for analysis (handling missing values, normalization, etc.).
  - Model Building: Use algorithms to create a model that can learn from the data.
  - Training: Feed the data into the model so it can learn.
  - Evaluation: Test the model with new data to see how well it performs.
  - Prediction: Use the model to make predictions on new data.

Types of Machine Learning:

- There are 3 main types of Machine Learning:
- **Supervised Learning:**
  - The machine is trained on labeled data, meaning each input has an associated correct output. The goal is for the machine to learn the mapping from input to output.
  - Example: Predicting the 3D structure of a protein from its sequence using known protein structures as training data.
  - Like a student learning with a teacher's guidance. The teacher provides the correct answers during practice.
- **Unsupervised Learning (Clustering):**
  - The machine is given data without explicit labels. It tries to find hidden patterns or structures in the data.
  - Example: Clustering patients based on gene expression profiles to discover new disease subtypes.
  - Like exploring a new city without a map, trying to discover the different neighborhoods on your own. Or Like a researcher discovering new protein families by grouping sequences with similar features.

Types of Machine Learning:

- **Reinforcement Learning:**
  - The machine learns by interacting with its environment, receiving rewards or penalties based on its actions. The goal is to maximize the cumulative reward.
  - The model learns through trial and error, receiving rewards for correct actions, which is less common but can be used in optimizing drug dosages.
  - Example:Optimizing a drug formulation process where the model learns the best conditions through iterative trials. Or A robot learning to navigate a maze by trial and error.
  - Like learning to ride a bicycle—falling a few times, but getting better with practice.

Using NumPy in ML for Bioinformatics:

- NumPy is extensively used in bioinformatics for handling large biological datasets, making it a cornerstone in data preprocessing and model building.
- Example in Bioinformatics:
  - Sequence Alignment: NumPy can be used to represent DNA sequences as matrices and perform operations like pairwise sequence alignment, essential for identifying homologous genes.
  - Protein Structure Prediction: Handle large datasets of protein structures and calculate distance matrices to predict the 3D structure using ML models.

Using NumPy in ML for Bioinformatics:

- What is NumPy?
  - NumPy (Numerical Python) is a library in Python that provides support for large multidimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays.
- .Installing NumPy using pip: `pip install numpu`
- Importing NumPy: To use NumPy, you must import it into your Python script
  - import numpy as np
- What is Array?
  - A NumPy array is a grid of values, all of the same type, indexed by a tuple of non-negative integers. Arrays are the central data structure of the NumPy library.
- Types of Array:
  - 1D Array: A single row of data: np.array([1, 2, 3])
  - 2D Array: A matrix (rows and columns): np.array([[1, 2], [3, 4]])
  - 3D Array: A collection of matrices: np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]]).

What is a Dimension?

- In the context of data science and machine learning, dimension refers to the structure or layout of data, typically described in terms of rows and columns.
- Rows: Each row represents a single observation or data point in your dataset. In bioinformatics, a row could represent an individual gene, protein, or a specific experiment.
- Columns: Each column represents a feature or attribute of the data. For example, in a dataset of genes, columns could represent the gene's expression level, function, or location in the genome.

Using NumPy in ML for Bioinformatics:

- **Array Types and Operations:**
  - **Zero-Dimensional Array (Scalar):**
    - Single value: np.array(42)
  - **High-Dimensional Arrays:**
    - Useful in advanced applications such as image processing where each pixel can be represented as an element in a multidimensional array.
  - **Row-Major vs Column-Major:**
    - NumPy uses row-major order, meaning rows are stored in contiguous memory locations. Understanding this helps in optimizing operations like matrix multiplication.

Using NumPy in ML for Bioinformatics:

- Setting Up: Installed and imported the necessary libraries.
- Data Understanding: Explored a bioinformatics dataset to understand the features and target variable.
- Data Preparation: Prepared the data by splitting it into training and testing sets and applied normalization using NumPy.
- Model Implementation: Used a logistic regression model to predict gene regulation.
- Evaluation: Evaluated the model's performance using accuracy as a metric.
- Advanced NumPy Concepts: Applied NumPy for normalization, array operations, and feature scaling.
- Visualization: Visualized the data and model predictions to understand the results.

Implementing a Basic ML Project:

- Project: Predicting drug-target interactions based on protein sequence data.
- Step 1: Data Collection
- Collect protein sequence data and known drug interactions from databases like DrugBank.
- Step 2: Data Preprocessing with NumPy
- Convert protein sequences into numerical representations (e.g., amino acid properties) using NumPy arrays.
- Normalize the data to ensure each protein's features are on a comparable scale.
- Step 3: Model Building
- Use a supervised learning algorithm like Support Vector Machines (SVM) to predict if a drug will bind to a target protein.
- Train the model on a dataset with known drug-protein interactions.
- Step 4: Model Evaluation
- Evaluate the model using precision, recall, and ROC-AUC to ensure it accurately predicts drug-protein interactions.
- Step 5: Prediction
- Use the model to predict the binding affinity of new drug compounds to a target protein, helping in drug repurposing or new drug discovery.

Data Preprocessing Techniques in Bioinformatics:

- Proper data preprocessing is critical in bioinformatics to ensure accurate predictions and discoveries:
  - **Normalization:** Scaling expression data (e.g., RNA-Seq data) to a common scale to compare gene expression levels across samples.
  - **Handling Missing Data:** Filling in missing values in protein interaction data using imputation techniques or by inferring from related data.
  - **Feature Engineering:** Creating new features such as chemical descriptors from molecular structures in drug design to enhance model accuracy.

Model Evaluation Metrics in Bioinformatics:

- When evaluating the performance of ML models in bioinformatics, specific metrics are used:
- Accuracy: Proportion of correct predictions, useful in disease diagnosis models.
- Precision and Recall: Important in drug discovery to evaluate how well the model identifies active compounds (precision) and how many active compounds are correctly identified (recall).
- F1 Score: Balances precision and recall, crucial in models where data is imbalanced, such as predicting rare disease mutations.
- ROC-AUC (Receiver Operating Characteristic - Area Under Curve): Commonly used in diagnosis models to measure the model's ability to distinguish between diseased and healthy patients.

Overfitting and Underfitting:

- Overfitting: When a model is too complex and captures noise in the training data, such as overfitting to specific patient data in a cancer diagnosis model, leading to poor generalization.
- Underfitting: When a model is too simple and fails to capture the underlying biology, such as a linear model failing to predict non-linear interactions in protein networks
- **Cross-Validation in Bioinformatic:**
  - .Cross-validation is vital in bioinformatics to ensure that models generalize well to unseen data, which is particularly important in clinical studies where sample sizes may be limited.
  - Example: Using k-fold cross-validation to evaluate the performance of a model predicting drug response, where the data is split into k subsets, and the model is trained and validated multiple times.

Ensemble Methods:

- Combining multiple models can improve prediction accuracy in complex bioinformatics tasks.
- Bagging: Like in Random Forest, where multiple decision trees predict protein functions based on sequence data, and their outputs are averaged to improve accuracy.
- Boosting: Models like Gradient Boosting are used to predict disease outcomes by sequentially correcting the errors of previous models, resulting in a robust predictor.

1. Data Collection

- Data collection is the first and one of the most crucial steps in any ML project. This involves gathering raw data that will be used to train and evaluate the machine learning model.
- **In Bioinformatics:** Data could be in the form of genomic sequences, protein structures, gene expression profiles, clinical trial data, or chemical compounds.
- **Example:** Suppose you are working on a project to predict protein-protein interactions. You might collect data from databases like STRING or BioGRID, where interaction data is available.
- **Challenges:** In bioinformatics, data collection might involve handling various formats (e.g., FASTA files for sequences, PDB files for structures) and dealing with data from multiple sources.

2. Data Preprocessing:

- Data preprocessing is essential to prepare the raw data for analysis. This step often involves cleaning the data, transforming it into a suitable format, and selecting relevant features.
- **In Bioinformatics:**
  - **Cleaning:** Removing or imputing missing values in gene expression data.
  - **Normalization:** Scaling numerical features like gene expression levels so they can be compared across different experiments.
  - **Feature Selection:** Selecting relevant features, such as specific amino acids in a protein sequence, that are known to be important for the function you're trying to predict.
- **Example:** If you have collected protein sequences, you might convert these sequences into numerical vectors based on amino acid properties, a process known as feature extraction.
- **Challenges:** Bioinformatics data can be noisy and incomplete. For example, in proteomics data, some proteins might have missing interaction partners, which you might need to infer or exclude.

3. Splitting the Data:

- Before building the model, the data is typically split into two or three sets: training, validation, and test sets.
- **Training Set:** Used to train the model. The model learns patterns and relationships from this data.
- Validation Set (optional): Used to tune the model's parameters (hyperparameters) and prevent overfitting.
- **Test Set:** Used to evaluate the model's performance on unseen data to ensure that the model generalizes well.
- **Example:** If you're predicting whether a protein will interact with another protein, you would use part of your data to train the model (training set) and another part to evaluate how well the model predicts interactions for new protein pairs (test set).
- **Challenges:** In bioinformatics, datasets can be imbalanced. For instance, in disease datasets, there might be far more non-disease samples than disease samples, which requires careful splitting to ensure the model isn't biased.

4. Model Selection:

- Model selection involves choosing the right algorithm or a combination of algorithms to solve your problem. The choice of model depends on the nature of the data and the specific problem you're trying to solve.
- **In Bioinformatics:**
  - **Classification Models:** Used for tasks like predicting whether a protein will be active/inactive (e.g., Support Vector Machines, Random Forest).
  - **Regression Models:** Used for predicting continuous outcomes, like the binding affinity of a drug to a protein (e.g., Linear Regression, Neural Networks).
  - **Clustering Models:** For grouping similar gene expression profiles (e.g., K-Means Clustering).
- **Example:** For predicting drug efficacy based on molecular features, you might choose a Random Forest model because it handles high-dimensional data well and can provide feature importance, helping you understand which molecular features are most predictive.
- **Challenges:** The complexity of biological data can make model selection challenging. Models that work well on simple data might not perform as well on complex, multi-dimensional bioinformatics data.

5. Model Training:

- Once the model is selected, it is trained using the training dataset. During training, the model learns the relationship between the input features (e.g., gene expression levels, protein sequences) and the output labels (e.g., disease/no disease, active/inactive).
- **In Bioinformatics:**
  - **Supervised Learning:** The model is trained on labeled data. For example, a model might be trained to classify a gene as either associated with a disease or not based on expression profiles.
  - **Unsupervised Learning:** The model is trained on unlabeled data, often to identify patterns or group similar data points. For example, clustering genes based on co-expression patterns.
- **Example:** Suppose you're building a model to predict protein function. You would feed in protein sequences and their known functions (from the training set) and let the model learn which sequence features are associated with each function.
- **Challenges:** Overfitting is a common problem where the model performs well on the training data but poorly on new, unseen data. Techniques like cross-validation and regularization are used to mitigate this.

6. Model Evaluation:

- After training, the model is evaluated to determine how well it performs on the test dataset. Evaluation metrics depend on the type of problem (classification, regression, etc.).
- **In Bioinformatics:**
  - **Accuracy:** For classification tasks, such as predicting if a gene is associated with a disease.
  - **Precision, Recall, and F1-Score:** Important when dealing with imbalanced datasets, such as predicting rare disease mutations.
  - **ROC-AUC:** Commonly used in bioinformatics to evaluate diagnostic models that predict disease presence.
- **Example:** After training a model to predict drug-target interactions, you might evaluate it by calculating the accuracy and ROC-AUC on a test set of drug-target pairs that were not used during training.
- **Challenges:** In bioinformatics, the heterogeneity of data can lead to different evaluation metrics showing varying results. For instance, a model might have high accuracy but low recall if it fails to predict many true positive interactions.

7. Model Tuning (Hyperparameter Optimization):

- Model tuning involves adjusting the model's hyperparameters to improve its performance. This step is often iterative and may involve techniques like grid search or random search.
- **In Bioinformatics:**
  - **Hyperparameters:** These include settings like the learning rate in neural networks or the number of trees in a Random Forest. Adjusting these can significantly impact model performance.
- **Example:** If you're using a Support Vector Machine (SVM) to classify proteins, you might tune the kernel type or regularization parameter to improve the model's accuracy.
- **Challenges:** Hyperparameter tuning can be computationally expensive, especially with large bioinformatics datasets. However, proper tuning is essential for maximizing the model's predictive power.

8. Model Deployment:

- Once the model is trained and evaluated, and its performance is satisfactory, it can be deployed to make predictions on new data.
- **In Bioinformatics:**
  - **Use Cases:** Predicting the efficacy of new drug compounds, identifying potential off-target effects of drugs, or diagnosing diseases based on patient gene expression profiles.
  - **Example:** Deploying a model in a clinical setting to predict a patient's response to a specific treatment based on their genomic data.
- Challenges: In bioinformatics, deploying models in real-world settings requires robust validation to ensure they perform well across diverse populations or conditions.

9. Monitoring and Updating the Model:

- After deployment, it is crucial to monitor the model's performance over time and update it as new data becomes available. This ensures the model remains accurate and relevant.
- **In Bioinformatics:**
  - **Continuous Learning:** As new biological data becomes available (e.g., new gene-disease associations), the model can be updated to incorporate this information.
  - **Example:** A model predicting drug interactions might need to be updated regularly as new drugs are developed and tested.
- Challenges: Biological data is continuously evolving. For example, new mutations might be discovered that affect disease diagnosis models, requiring updates to the model to maintain accuracy.

Thank-You!