Python Programming
MCA-106

UNIT-I
Introduction to Python

Learning Objectives

In this unit, we'll cover the following:

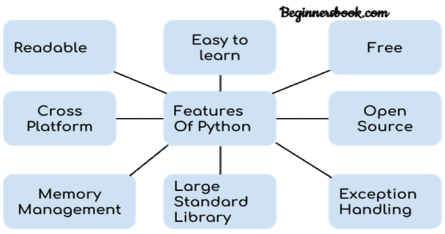
- History
- Features and Future of Python
- Installation
- Interactive Shell and Program Structure
- Identifiers, Keywords, Variables, Assignments, Immutable Variables Escape Sequences
- Data-Types, Operators and Operands,
- Command-Line arguments
- Control Flow
- Functions
- Modules

History

- Invented in the Netherlands, early 90s by Guido van Rossum
- Named after Monty Python
- Open sourced from the beginning
- Considered a scripting language, but is much more scalable, object oriented and functional from the beginning
- Used by Google from the beginning
- Increasingly popular
- Python 2.0 was released on 16 October 2000
- Python 3.0 was released on 3 December 2008 (Latest is 3.9.4)

 4

Features of Python



 5

Features of Python continued …

1. **Readable:** Python is a very readable language.
2. **Easy to Learn:** Python is a high level programming (OOP) language still it is easy to understand and learn the language
3. **Cross platform:** Python is available and can run on various operating systems such as Mac, Windows, Linux, Unix etc. This makes it a cross platform and portable language.
4. **Open Source:** Python is a open source programming language.
5. **Large standard library:** Python comes with a large standard library that has some handy codes and functions which we can use while writing code in Python.

 6

Features of Python continued …

6. **Free:** Python is free to download and use. This means you can download it for free and use it in your application. See: Open Source Python License.
7. Python is an example of a FLOSS (Free/Libre Open Source Software), which means you can freely distribute copies of this software, read its source code and even modify it.
8. **Supports exception handling**
9. **Advanced features:** Supports Dynamic typing, Programming-in-the large support, Built-in object types, Built-in tools ,Library utilities, Third-party utilities
10. **Automatic memory management:** Python supports automatic memory management which means the memory is cleared and freed automatically. You do not have to bother clearing the memory.

 7

Applications of Python

- Web development
- Machine learning / Mathematics
- Data Analysis
- Scripting
- Game development
- Embedded applications development
- Desktop applications
- System Programming

 8

Installation of Python

- You can install Python on any operating system such as Windows, Mac OS X, Linux/Unix and others.
- To install the Python on your operating system, go to this link: https://www.python.org/downloads/
- Download latest version 3.9.4
- Installation steps are simple. You just have to accept the agreement and finish the installation.
- Python IDEs and code editors – IDLE, PyCharm, Visual Studio Code, Spyder, Anaconda (Data Science- Python & R for scientific programming)
- Three primary implementations of the Python language—CPython, Jython, and IronPython

 9

Interactive Shell

We will be using IDLE shell 3.9.4

- IDLE is an Integrated DeveLopment Environment for Python, typically used on Windows
- Multi-window text editor with syntax highlighting, auto-completion, smart indent and other.
- Python shell with syntax highlighting.
- Integrated debugger with stepping, persistent breakpoints, and call stack visibility

 10

Program Structure and Execution

- Python was designed for readability
- Python uses new lines to complete a command, as opposed to other programming languages
- Python Indentation - Python uses indentation to indicate a block of code and define scope of loops, functions and classes
- Python Comments (#)
- Multi-line comments (“ ” ” – Triple quotes)

 11

Program Structure and Execution

- Programmer’s view - a Python program is just a text file containing Python statements.
- Python’s view - It’s first compiled to something called “byte code” and then routed to something called a “virtual machine.”
- Source code(.py) → Bite Code (.pyc) → Python Virtual Machine (PVM)

 12

First Python Program

```
# This program adds two numbers

num1 = 1.5
num2 = 6.3

# Add two numbers
sum = num1 + num2

# Display the sum
print('The sum of {0} and {1} is {2}'.format(num1, num2, sum))
```


First Python Program

```
# Store input numbers
num1 = input('Enter first number: ')
num2 = input('Enter second number: ')

# Add two numbers
sum = float(num1) + float(num2)

# Display the sum
print('The sum of {0} and {1} is {2}'.format(num1, num2, sum))
```


Python Variable

Variables are containers for storing data values.

- Python has no command for declaring a variable.
- A variable is created the moment you first assign a value to it.
- Variables do not need to be declared with any particular *type*, and can even change type after they have been set.

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

- If you want to specify the data type of a variable, this can be done with casting.

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

- You can get the data type of a variable with the type() function

Python Variable continued …

String variables can be declared either by using single or double quotes

```
x = “Arjun"
# is the same as
x = ‘Arjun‘
```

Variable names are case-sensitive.

A variable can have a short name (like x and y) or a more descriptive name (age, carname, total_volume). Rules for Python variables:

- A variable name must start with a letter or the underscore character
- A variable name cannot start with a number
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
- Variable names are case-sensitive (age, Age and AGE are three different variables)

Python Variable continued …

Python allows you to assign values to multiple variables in one line:

```
x, y, z = “RED", “GREEN", “BLUE“
```

And you can assign the *same* value to multiple variables in one line:

```
x = y = z = “RED“
```

If you have a collection of values in a list, tuple etc. Python allows you extract the values into variables. This is called ***unpacking.***

```
colors= ["RED",  "GREEN", "BLUE"]
x, y, z = colors
print(x)
print(y)
print(z)
```


Python Variable continued …

Python - Output Variables

The python ***print*** statement is often used to output variables. To combine both text and a variable, Python uses the **+** character.

```
x = “Powerful"
print("Python is " + x + “ language”)
```

Python - Global Variables

- Variables that are created outside of a function (as in all of the examples above) are known as global variables.
- Global variables can be used by everyone, both inside of functions and outside.

Python Data Types

Built-in Data Types

- In programming, data type is an important concept.
- Variables can store data of different types, and different types can do different things.
- Python has the following data types built-in by default, in these categories:

Data Type	Keyword used in Python
Text Type	str
Numeric Type	int, float, complex
Sequence Type	list, tuple, range
Mapping Type	Dict
Set Type	set, frozenset
Boolean Type	bool
Binary Types	bytes, bytearray, memoryview

 19

Python String

Strings in python are surrounded by either single quotation marks, or double quotation marks.

'Powerful' is same as "Powerful"

You can display a string literal with the ***print()*** function.

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

a = "Hello" or a = 'Hello'

You can assign a multiline string to a variable by using three quotes:

a = " " " Python was designed for readability, and has some similarities to the English language with influence from mathematics. " " "

Or

a = ' ' ' Python was designed for readability, and has some similarities to the English language with influence from mathematics. ' ' '

 20

Python String continued…

Strings are Arrays

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

Get the character at position 1 (remember that the first character has the position 0):

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

 21

Python String continued…

String functions or operations on strings:

- Looping Through a String

```
for x in “BVICAM":
    print(x)
```

- String Length – len()
- Check String – returns boolean

```
txt = "The best things in life are free!"
print("free" in txt)
```

- Check if NOT - returns boolean

```
txt = "The best things in life are free!"
print("expensive" not in txt)
```

- Slicing

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


Python String continued…

Upper Case

Lower Case

Remove whitespace – strip()

Replace String

Split String

String Concatenation

String Format - we can combine strings and numbers by using the format method. The format() method takes the passed arguments, formats them, and places them in the string where the placeholders { } are:

```
code= 106
txt = “Paper Code of Python Programming is MCA - { }"
print(txt.format(code))
```


Python Escape Characters

To insert characters that are illegal in a string, use an escape character.

An escape character is a backslash \ followed by the character you want to insert.

An example of an illegal character is a single or double quote inside a string that is surrounded by double quotes:

```
txt = ‘MCA Students\’ Datasheet’
```

Python Operators

Operators are used to perform operations on variables and values

Python divides the operators in the following groups:

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

Python Operators continued ..

Arithmetic operators

Operator	Description	Example
+ Addition	Adds values on either side of the operator.	a + b = 30
- Subtraction	Subtracts right hand operand from left hand operand.	a – b = -10
* Multiplication	Multiplies values on either side of the operator	a * b = 200
/ Division	Divides left hand operand by right hand operand	b / a = 2
% Modulus	Divides left hand operand by right hand operand and returns remainder	b % a = 0
** Exponent	Performs exponential (power) calculation on operators	a**b =10 to the power 20
//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed. But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity) –	9//2 = 4 and 9.0//2.0 = 4.0, -11//3 = -4, -11.0//3

Python Operators

Assignment operators

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

Python Operators

Comparison operators

Operator	Description	Example
==	If the values of two operands are equal, then the condition becomes true.	(a == b) is not true.
!=	If values of two operands are not equal, then condition becomes true.	(a != b) is true.
<>	If values of two operands are not equal, then condition becomes true.	(a <> b) is true. This is similar to != operator.
>	If the value of left operand is greater than the value of right operand, then condition becomes true.	(a > b) is not true.
<	If the value of left operand is less than the value of right operand, then condition becomes true.	(a < b) is true.
>=	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.	(a >= b) is not true.
<=	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.	(a <= b) is true.

Python Operators

Logical operators

Operator	Description	Example
and Logical AND	If both the operands are true then condition becomes true.	(a and b) is true.
or Logical OR	If any of the two operands are non-zero then condition becomes true.	(a or b) is true.
not Logical NOT	Used to reverse the logical state of its operand.	Not(a and b) is false.

Python Operators

Identity operators

Operator	Description	Example
is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.	x is y, here **is** results in 1 if id(x) equals id(y).
is not	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.	x is not y, here **is not** results in 1 if id(x) is not equal to id(y).

x=3
y=x
x is y

a=3
b=5
a is b

Python Operators

Membership operators

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.

```
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"
```

Now set a=2 and try the above code

Python Operators

Bitwise operators – Consider a = 0011 1100 and b = 0000 1101

Operator	Description	Example
& Binary AND	Operator copies a bit to the result if it exists in both operands	(a & b) (means 0000 1100)
| Binary OR	It copies a bit if it exists in either operand.	(a | b) = 61 (means 0011 1101)
^ Binary XOR	It copies the bit if it is set in one operand but not both.	(a ^ b) = 49 (means 0011 0001)
~ Binary Ones Complement	It is unary and has the effect of 'flipping' bits.	(~a) = -1100 0011
<< Binary Left Shift	The left operands value is moved left by the number of bits specified by the right operand.	a << 2 = 240 (means 1111 0000)
>> Binary Right Shift	The left operands value is moved right by the number of bits specified by the right operand.	a >> 2 = 15 (means 0000 1111)

Operators Precedence

Sr.No.	Operator & Description
1	****** Exponentiation (raise to the power)
2	**~ + -** Complement, unary plus and minus (method names for the last two are +@ and -@)
3	*** / % //** Multiply, divide, modulo and floor division
4	**+ -** Addition and subtraction
5	**>> <<** Right and left bitwise shift
6	**&** Bitwise 'AND'
7	**^ |** Bitwise exclusive 'OR' and regular 'OR'
8	**<= < > >=** Comparison operators
9	**<> == !=** Equality operators
10	**= %= /= //= -= += *= **=** Assignment operators
11	**is or is not** Identity operators
12	**in not in** Membership operators
13	**not or and** Logical operators

Type conversion in Python

Python defines type conversion functions to directly convert one data type to another which is useful in day to day and competitive programming.

There are two types of Type Conversion in Python:

- ***Implicit Type Conversion***

  In Implicit type conversion of data types in Python, the Python interpreter automatically converts one data type to another without any user involvement.

```
x = 10
print("x is of type:",type(x))
y = 10.6
print("y is of type:",type(y))
x = x + y
print(x)
print("x is of type:",type(x))
```


Type conversion in Python

- ***Explicit Type Conversion***

  In Explicit Type Conversion in Python, the data type is manually changed by the user as per their requirement.

```
#convert from int to float:
x = float(1)
#convert from float to int:
y = int(2.8)
#convert from int to complex:
z = complex(x)
print(x)
print(y)
print(z)

print(type(x))
print(type(y))
print(type(z))
```


Short Circuit Evaluation in Python

Short-circuit evaluation means that when evaluating Boolean expression like AND and OR, you can stop as soon as you find the first condition which satisfies or negates the expression.

Operation	Result	Description
x or y	If x is false, then y else x	Only evaluates the second argument(y) if the firs one is false
x and y	If x is false, then x else y	Only evaluates the second argument(y) if the first one(x) is True
not x	If x is false, then True, else False	Not has a lower priority than non-boolean operators

Lazy Evaluation in Python

Lazy evaluation is an evaluation strategy which holds the evaluation of an expression until its value is needed. It avoids repeated evaluation.

Lazy Evaluation − Advantages

- It allows the language runtime to discard sub-expressions that are not directly linked to the final result of the expression.
- It reduces the time complexity of an algorithm by discarding the temporary computations and conditionals.
- It allows the programmer to access components of data structures out-of-order after initializing them, as long as they are free from any circular dependencies.
- It is best suited for loading data which will be infrequently accessed.

Lazy Evaluation in Python

Lazy Evaluation − Drawbacks

- It forces the language runtime to hold the evaluation of sub-expressions until it is required in the final result by creating **thunks** (delayed objects).
- Sometimes it increases space complexity of an algorithm.
- It is very difficult to find its performance because it contains thunks of expressions before their execution.

Lazy Evaluation in Python

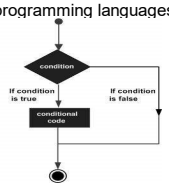
- The **range** method in Python follows the concept of Lazy Evaluation. It returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and stops before a specified number.

```
r = range(start, stop, step)
r= range(6)
r= range(3,6)
r= range(3,20,2)
```


Conditional Statements in Python

Decision making is anticipation of conditions occurring while execution of the program and specifying actions taken according to the conditions.

- 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 −



Conditional Statements in Python

- Python programming language assumes any **non-zero** and **non-null** values as TRUE, and if it is either **zero** or **null**, then it is assumed as FALSE value.
- Python programming language provides following types of decision making statements. Click the following links to check their detail.

Sr.No.	Statement & Description
1	if statements An **if statement** consists of a boolean expression followed by one or more statements.
2	if...else statements An **if statement** can be followed by an optional **else statement**, which executes when the boolean expression is FALSE.
3	nested if statements You can use one **if** or **else if** statement inside another **if** or **else if** statement(s).

 40

Conditional Statements in Python

Python Conditions and If statements

Python supports the usual logical conditions from mathematics:

- Equals: a == b
- Not Equals: a != b
- Less than: a < b
- Less than or equal to: a <= b
- Greater than: a > b
- Greater than or equal to: a >= b

```
a = 33
b = 200
if b > a:
  print("b is greater than a")
elif a== b:
  print("a and b are equal")
else:
  print("a is greater than b")
```

 41

Conditional Statements in Python

Short Hand If

If you have only one statement to execute, you can put it on the same line as the if statement : Example

```
if a > b: print("a is greater than b")
```

Short Hand If ... Else

- If you have only one statement to execute, one for if, and one for else, you can put it all on the same line: Example

```
a = 2
b = 330
print("A") if a > b else print("B")
```

The pass Statement

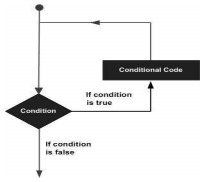
If statements cannot be empty, but if you for some reason have an if statement with no content, put in the statement to avoid getting an error.

```
if b > a:
  pass
```

 42

Python Loops

- In general, statements are executed sequentially: But, there may be a situation when you need to execute a block of code several number of times.
- Programming languages provide various control structures that allow for more complicated execution paths.
- A loop statement allows us to execute a statement or group of statements multiple times. The following diagram illustrates a loop statement −



 43

Python Loops

Python programming language provides following types of loops to handle looping requirements.

Sr.No.	Loop Type & Description
1	while loop Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body.
2	for loop Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
3	nested loops You can use one or more loop inside any another while, for or do..while loop.

 44

Python Loops

- Loop Control Statements
- Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.
- Python supports the following control statements. Click the following links to check their detail.
- Let us go through the loop control statements briefly

Sr.No.	Control Statement & Description
1	break statement Terminates the loop statement and transfers execution to the statement immediately following the loop.
2	continue statement Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
3	pass statement The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.

 45

Python Loops

Python while Loop Statements

- A **while** loop statement in Python programming language repeatedly executes a target statement as long as a given condition is true.
- The syntax of a **while** loop in Python programming language is −

```
while expression
        statements(s)
```

- Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression, and true is any non-zero value. The loop iterates while the condition is true.
- When the condition becomes false, program control passes to the line immediately following the loop.
- In Python, all the statements indented by the same number of character spaces after a programming construct are considered to be part of a single block of code. Python uses indentation as its method of grouping statements.

Python Loops

Using else Statement with While Loop

- Python supports to have an **else** statement associated with a loop statement.
- If the **else** statement is used with a **while** loop, the **else** statement is executed when the condition becomes false.
- The following example illustrates the combination of an else statement with a while statement that prints a number as long as it is less than 5, otherwise else statement gets executed.

```
count = 0
while count < 5:
        print count, " is less than 5 "
        count = count +1
else:
        print count, " is not less than 5 "
```


Python Loops

Python for Loop Statements

It has the ability to iterate over the items of any sequence, such as a list or a string.

```
        for iterating_var in sequence:
                statement(s)
```

Example:

```
        for letter in "BVICAM"
                print 'Current Letter:' , letter

        colors = ['RED', 'GREEN', 'BLUE']
        for color in colors:
                print 'Current Color:' , color
```

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.

Basically, we can divide functions into the following two types:

- **Built-in functions** - Functions that are built into Python.
- For example – abs(), complex(), dict(), float(), format(), id()

- **User-defined functions** - Functions defined by the users themselves.

 49

User-defined functions

Advantages of user-defined functions

- User-defined functions help to decompose a large program into small segments which makes program easy to understand, maintain and debug. (Modular Programming)
- If repeated code occurs in a program, functions can be used to include those codes and execute when needed by calling that function.
- Programmers working on large project can divide the workload by making different functions.

 50

User-defined functions

Defining a *user-defined functions*

- Function blocks begin with the keyword **def** followed by the function name and parentheses (( )).
- Any input parameters or arguments should be placed within these parentheses.
- The first statement of a function can be an optional statement - the documentation string of the function or *docstring*.
- The code block within every function starts with a colon (:) and is indented.
- The statement return [expression] exits a function. A return statement with no arguments is the same as return None.
- **pass** statement can be used in case function has an empty body.

 51

Functions

```
def function_name( paramenters ) :
        "Function_docstring"
        Fucntion_Body
        return [expression]

Example :
def print_me( str ):
        "This prints a passed string into this function"
        print str
        return

print_me("Test String")
```


Arguments

Arguments

Information can be passed into functions as arguments.

You can add as many arguments as you want, just separate them with a comma.

Number of Arguments

By default, a function must be called with the correct number of arguments.

Arbitrary Arguments, *args

- If you do not know how many arguments that will be passed into your function, add a * before the parameter name in the function definition.
- This way the function will receive a *tuple* of arguments, and can access the items accordingly

```
def my_function(*students)
        print("The topper of the class is " + students[3])
```


Arguments

Keyword Arguments

- You can also send arguments with the *key* = *value* syntax.
- This way the order of the arguments does not matter.

```
def my_function(stud1, stud2, stud3, stud4)
        print("The topper of the class is " + students[3])

my_function(stud2='Amit', stud3='Suman', stud1= 'Nikita', stud4='Parth')
```

Default Parameter Value

```
def my_function(country = 'INDIA')
        print("I am from " + country)
my_fucntion("Canada")
my_fucntion ()
```

Scope and Lifetime of variables

- Scope of a variable is the portion of a program where the variable is recognized. Parameters and variables defined inside a function are not visible from outside the function. Hence, they have a local scope.
- The lifetime of a variable is the period throughout which the variable exits in the memory. The lifetime of variables inside a function is as long as the function executes.
- They are destroyed once we return from the function. Hence, a function does not remember the value of a variable from its previous calls.

```
def my_function():
        x=10
        print("Value of x inside function = " , x)
my_function()
x=20
print("Value of x outside function = " , x)
```

 55

Scope and Lifetime of variables

Global vs. Local variables

- Variables that are defined inside a function body have a local scope, and those defined outside have a global scope.
- This means that local variables can be accessed only inside the function in which they are declared, whereas global variables can be accessed throughout the program body by all functions. When you call a function, the variables declared inside it are brought into scope.

 56

Calling a Function

- Defining a function only gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code.
- Once the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt.

Pass by reference vs value

All parameters (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function.

```
def change_me(mylist):
        mylist.append([1,2,3,4])
        print "value inside the function: " , mylist
        return
mylist =[10,20,30]
change_me( mylist )
print "value outside the function: " , mylist
```

 57

Calling a Function

Example:

```
def my_function(x):
        x[0] = 20
        return

# After function call
my_list= [10,11,12,13,14,15]
my_function(my_list)
print (my_list)
```


Calling a Function

When we pass a reference and change the received reference to something else, the connection between the passed and received parameter is broken.

```
def my_function(x):
        x = [20, 30, 40]
        print (my_list)
        return

# After function call
my_list= [10,11,12,13,14,15]
my_function(my_list)
print (my_list)
```


The *Anonymous* Functions

These functions are called anonymous because they are not declared in the standard manner by using the *def* keyword. You can use the *lambda* keyword to create small anonymous functions.

- Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.
- An anonymous function cannot be a direct call to print because lambda requires an expression
- Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.
- Although it appears that lambda's are a one-line version of a function, they are not equivalent to inline statements in C or C++, whose purpose is by passing function stack allocation during invocation for performance reasons.

Syntax

The syntax of *lambda* functions contains only a single statement, which is as follows −

```
        Lambda [ agr1, [arg2, ..... agrn ]]: expression
Sum = lambda agr1, agr2 : arg1 +arg2
Print "Value of total = " , sum(10,10)
Print "Value of total = " , sum(20,20)
```

Rercursion

Recursion is the process of defining something in terms of itself.

In Python, we know that a function can call other functions. It is even possible for the function to call itself. These types of construct are termed as recursive functions.

Advantages of using recursion

- A complicated function can be split down into smaller sub-problems utilizing recursion.
- Sequence creation is simpler through recursion than utilizing any nested iteration.
- Recursive functions render the code look simple and effective.

Disadvantages of using recursion

- A lot of memory and time is taken through recursive calls which makes it expensive for use.
- Recursive functions are challenging to debug.
- The reasoning behind recursion can sometimes be tough to think through.

Rercursion

Syntax:

```
def recurse():
    ...
    recurse()
    ...

recurse()
```

recursive call

Example:

```
def factorial(x):
        if x==1 :
                Return 1
        else:
                return(x * factorial(x-1))
num = 3
print("The factorial of ", num, " is " , factorial(num))
```


Rercursion

- The developer should be very careful with recursion as it can be quite easy to slip into writing a function which never terminates, or one that uses excess amounts of memory or processor power.
- However, when written correctly recursion can be a very efficient and mathematically-elegant approach to programming.
- To a new developer it can take some time to work out how exactly this works, best way to find out is by testing and modifying it.
- Every recursive function must have a base condition that stops the recursion or else the function calls itself infinitely.
- The Python interpreter limits the depths of recursion to help avoid infinite recursions, resulting in stack overflows.
- By default, the maximum depth of recursion is 1000. If the limit is crossed, it results in RecursionError.

```
def recursor():
    recursor()
recursor()
```

Modules

Python **functions,** Python **modules** and Python **packages**, three mechanisms that facilitate **modular programming**.

Modular programming refers to the process of breaking a large programming task into separate, smaller, more manageable subtasks or **modules**. Individual modules can then be cobbled together like building blocks to create a larger application. several advantages to **modularizing** code in a large application:

- **Simplicity:** Rather than focusing on the entire problem at hand, a module typically focuses on one relatively small portion of the problem.
- **Maintainability:** Modules are typically designed so that they enforce logical boundaries between different problem domains. If modules are written in a way that minimizes interdependency, there is decreased likelihood that modifications to a single module will have an impact on other parts of the program.
- **Reusability:** Functionality defined in a single module can be easily reused
- **Scoping:** Modules typically define a separate **namespace**, which helps avoid collisions between identifiers in different areas of a program.

Modules

In programming, a module is a piece of software that has a specific functionality.

- Modules in Python are simply Python files with a .py extension. A file containing a set of functions you want to include in your application.
- The name of the module will be the name of the file.
- A Python module can have a set of functions, classes or variables defined and implemented.

```
def greeting(name):
    print("Hello " + name)
```

- Now we can use the module we just created, by using the ***import*** statement

```
import mymodule
mymodule.greeting("Aarti")
```

- You can create an alias when you import a module, by using the ***as*** keyword

```
import mymodule as mx
mx.greeting("Aarti")
```


Exploring built-in Modules

There is a huge list of built-in modules in the Python standard library.

Two very important functions come in handy when exploring modules in Python – the ***dir*** and ***help*** functions.

Built-in modules are written in C and integrated with the Python shell. Each built-in module contains resources for certain system-specific functionalities such as OS management, disk IO, etc. The standard library also contains many Python scripts (with the .py extension) containing useful utilities.

To display a list of all available modules, use the following command in the Python console:

```
>>> help('modules')
```

There is a built-in function to list all the function names (or variable names) in a module. The ***dir()*** function:

```
import platform
x = dir(platform)
print(x)
```

Math Module

Python has a built-in module that you can use for mathematical tasks. It is a standard module in Python and is always available.

To use mathematical functions under this module, you have to import the module using

import math

Functions in Python Math Module

math.ceil()
math.exp()
math.pow()
math.sqrt()
math.fmod()
math.fabs()
math.factorial()
math.gcd()

Math Constants

math.e
math.pi

Random Module

Python has a built-in module that you can use to make random numbers.

Method	Description
seed()	Initialize the random number generator
shuffle()	Takes a sequence and returns the sequence in a random order
sample()	Returns a given sample of a sequence
random()	Returns a random float number between 0 and 1
randrange()	Returns a random number between the given range
uniform()	Returns a random float number between two given parameters

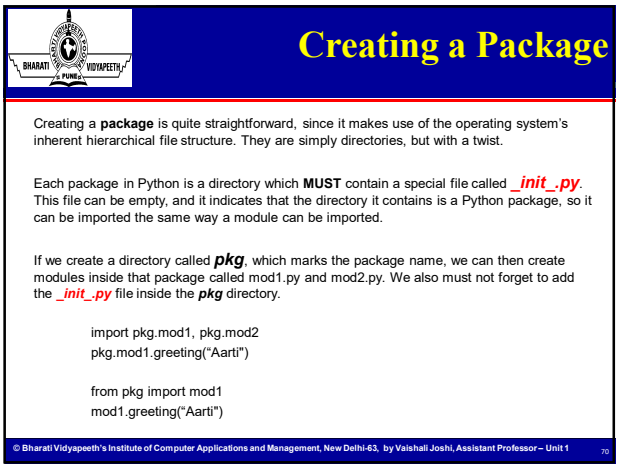
Python Packages

As an application program grows larger in size , it includes a lot of modules.

As the number of modules grows, it becomes difficult to keep track of them all if they are dumped into one location. This is particularly so if they have similar names or functionality. You might wish for a means of grouping and organizing them.

Packages allow for a hierarchical structuring of the module namespace using **dot notation**. In the same way that **modules** help avoid collisions between global variable names, **packages** help avoid collisions between module names.

Packages are analogous to directories and modules for files. As a directory can contain subdirectories and files, a Python package can have sub-packages and modules.


Creating a Package
Creating a package is quite straightforward, since it makes use of the operating system's inherent hierarchical file structure. They are simply directories, but with a twist.
Each package in Python is a directory which MUST contain a special file called _init_.py. This file can be empty, and it indicates that the directory it contains is a Python package, so it can be imported the same way a module can be imported.
If we create a directory called pkg, which marks the package name, we can then create modules inside that package called mod1.py and mod2.py. We also must not forget to add the _init_.py file inside the pkg directory.
import pkg.mod1, pkg.mod2
pkg.mod1.greeting("Aarti")
from pkg import mod1
mod1.greeting("Aarti")
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Vaishali Joshi, Assistant Professor – Unit 1
70

UNIT-II

Learning Objectives

In this unit, we'll cover the following:

Python Data Types

Built-in Data Types

- In programming, data type is an important concept.
- Variables can store data of different types, and different types can do different things.
- Python has the following data types built-in by default, in these categories:

Data Type	Keyword used in Python
Text Type	str
Numeric Type	int, float, complex
Sequence Type	list, tuple, range
Mapping Type	Dict
Set Type	set, frozenset
Boolean Type	bool
Binary Types	bytes, bytearray, memoryview

Python String

Strings in python are surrounded by either single quotation marks, or double quotation marks.

'Powerful' is same as "Powerful"

You can display a string literal with the ***print()*** function.

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

a = "Hello" or a = 'Hello'

You can assign a multiline string to a variable by using three quotes:

a = " " " Python was designed for readability, and has some similarities to the English language with influence from mathematics. " " "

Or

a = ' ' ' Python was designed for readability, and has some similarities to the English language with influence from mathematics. ' ' '

Python String continued…

Strings are Arrays

- Like many other popular programming languages, strings in Python are arrays of bytes representing unicode characters.
- However, Python does not have a character data type, a single character is simply a string with a length of 1.
- Square brackets can be used to access elements of the string.
- String data types are immutable. Which means a string value cannot be updated.

Get the character at position 1 (remember that the first character has the position 0):

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


Python String continued…

String functions or operations on strings:

- Looping Through a String

```
for x in "BVICAM":
    print(x)
```

- String Length – len()
- Check String – returns boolean

```
txt = "The best things in life are free!"
print("free" in txt)
```

- Check if NOT - returns boolean

```
txt = "The best things in life are free!"
print("expensive" not in txt)
```

- Slicing

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

Python String continued…

Upper Case – upper()
Lower Case – lower()
Remove whitespace – strip(), lstrip(), rstrip()
Replace String – str.replace(old_str, new_str)
Split String – str.split()
String Concatenation – plus(+) operator
Deleting string – del str

String Format - we can combine strings and numbers by using the format method. The format() method takes the passed arguments, formats them, and places them in the string where the placeholders { } are:

```
code= 106
txt = "Paper Code of Python Programming is MCA - { }"
print(txt.format(code))
```


Sequence Type

Sequences allow you to store multiple values in an organized and efficient manner. There are several **sequence types**: strings, lists, tuples, bytearrays, and range objects. Two most popular sequence types are **lists** and **Tuples**.

- Each element of a sequence is assigned a number - its position or index. The first index is zero, the second index is one, and so forth.
- Operations that can be performed on sequence types include indexing, slicing, adding, multiplying, and checking for membership.
- In addition, Python has built-in functions for finding the length of a sequence and for finding its largest and smallest elements.

Python Lists

Python lists

The list is written as a list of comma-separated values (items) between square brackets.

Important thing about a list is that items in a list need not be of the same type.

Creating a list is as simple as putting different comma-separated values between square brackets

```
list1 = ['physics', 'chemistry', 1997, 2000 ];
list2 = [1, 2, 3, 4, 5, 6,7,8 ]
list3 = ["a", "b", "c", "d" ]
print "list1[0]: ", list1[0]
print "list2[1:5]: ", list2[1:5]
list1[2] = 2021
del list1[2]
```

Python Lists continued..

- List items are **ordered**, **changeable**, and allow **duplicate** values.
- When we say that lists are ordered, it means that the items have a defined order, and that order will not change. If you add new items to a list, the new items will be placed at the end of the list.
- The list is changeable, meaning that we can change, add, and remove items in a list after it has been created.
- Since lists are indexed, lists can have items with the same value
- To determine how many items a list has, use the len() function

  print(len(list1))
- 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.

  del list1[2]

  List1.remove('chemistry')

Python Lists Methods

Sr.No.	Methods with Description
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
10	list.index(element, start, end) returns the index of the specified element in the list.

Cloning a list in Python

There are various ways of copying or cloning a list in Python. These various ways of copying takes different execution time, so we can compare them on the basis of time.

- **Using slicing technique**
  This is the easiest and the fastest way to clone a list. This method is considered when we want to modify a list and also keep a copy of the original. This process is also called cloning. This technique takes about 0.039 seconds and is the fastest technique.

  list2=list1[:]
- **Using the extend() method**
  The lists can be copied into a new list by using the extend() function. This appends each element of the iterable object (e.g., anothre list) to the end of the new list. This takes around 0.053 second to complete.

  list2.extend(list1)

Cloning a list in Python

- **Using the list() method**
  This is the simplest method of cloning a list by using the builtin function list(). This takes about 0.075 seconds to complete.

```
list2 = list(list1)
```

- **Using the copy() method**
  The inbuilt method copy is used to copy all the elements from one list to another. This takes around 1.488 seconds to complete.

```
list2= list1.copy()
```

- **Using list comprehension**
  The method of list comprehension can be used to copy all the elements individually from one list to another. This takes around 0.217 seconds to complete.

```
list2 = [i for I in list1]
```


Cloning a list in Python

- **Using the append() method**
  This can be used for appending and adding elements to list or copying them to a new list. It is used to add elements to the last position of list. This takes around 0.325 seconds to complete and is the slowest method of cloning.

```
list2=[]
for item in list1: list2.append(item)
```


Python Tuples

Python Tuples

- A tuple is a collection of objects which ordered and immutable.
- Tuples are sequences, just like lists. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets.
- Creating a tuple is as simple as putting different comma-separated values.

```
tup1= ('physics', 'chemistry', 1997, 2000 )
tup2 = (1, 2, 3, 4, 5, 6,7,8 )
tup3 = "a", "b", "c", "d"
tup4 =(50,)
print "tup1[0]: ", tup1[0]
print "tup2[1:5]: ", tup2[1:5]
```

- Please note, **Tuples are immutable** which means you cannot update or change the values of tuple elements. You are able to take portions of existing tuples to create new tuples

Python Tuples

Tuple Assignment, Packing, and Unpacking

A literal tuple containing several items can be assigned to a single object:

```
tup1= ('physics', 'chemistry', 1997, 2000 )
```

When this occurs, it is as though the items in the tuple have been "packed" into the object.

If that "packed" object is subsequently assigned to a new tuple, the individual items are "unpacked" into the objects in the tuple:

```
(s1,s2,s3,s4) = tup1
```

When unpacking, the number of variables on the left must match the number of values in the tuple

Create Tuple With One Item

- To create a tuple with only one item, you have to add a comma after the item, otherwise Python will not recognize it as a tuple.

Python Tuples

- **Advantages of Tuple over List**
- Since tuples are quite similar to lists, both of them are used in similar situations. However, there are certain advantages of implementing a tuple over a list. Below listed are some of the main advantages:
- We generally use tuples for heterogeneous (different) data types and lists for homogeneous (similar) data types.
- Since tuples are immutable, iterating through a tuple is faster than with list. So there is a slight performance boost.
- Tuples that contain immutable elements can be used as a key for a dictionary. With lists, this is not possible.
- If you have data that doesn't change, implementing it as tuple will guarantee that it remains write-protected.

Python Mapping Type

Python Dictionary

- Dictionaries are used to store data values in key:value pairs.
- A dictionary is a collection which is ordered*, changeable and does not allow duplicates.
- Dictionaries are written with curly brackets, and have keys and values:
- Each key is separated from its value by a colon (:), the items are separated by commas, and the whole thing is enclosed in curly braces.
- Keys are unique within a dictionary while values may not be.
- The values of a dictionary can be of any type, but the keys must be of an immutable data type such as strings, numbers, or tuples.

```
dict = {
'Name': 'Arjun',
'Age': 21,
'Course': 'MCA'}
Print(dict["Name"])
```

Python Dictionary continued…

Accessing Items

```
x = dict.get("Name")
```

Change Values

```
dict.update({"Course" : "MBA"})
```

Adding Items

```
dict.update({"City" : "New Delhi"})
```

Removing Items

```
del dict ["City"]
```

Loop through a Dictionary:

- Print all key names in the dictionary, one by one:

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

- Print all *values* in the dictionary, one by one:

```
for x in dict:
    print(dict[x])
```


Python Dictionary continued…

Copy a Dictionary

You cannot copy a dictionary simply by typing dict1 = dict2

- One way is to use the built-in Dictionary method ***copy()***

```
dict1 = { 'Name': 'Arjun', 'Age': 21, 'Course': 'MCA'}
dict2 = dict2.copy()
print(dict2)
```

- Another way to make a copy is to use the built-in function ***dict()***

```
dict1 = { 'Name': 'Arjun', 'Age': 21, 'Course': 'MCA'}
dict2 = dict(dict2)
print(dict2)
```


Python Dictionary continued…

Nested Dictionaries

A dictionary can contain dictionaries, this is called nested dictionaries.

```
My_Library ={
"Book1": {
        'Title': 'Fundamentals of IT',
        'Author':'Kapoor',
        'Publisher':'TMH',
        'Year': 2006
        },
"Book2": {
        'Title': 'Introduction to Python Programming',
        'Author':'Timothy A. Budd',
        'Publisher': 'S.Chand',
        'Year': 2016
        },
"Book3": {
        'Title': 'PL/'SQL Programming,
        'Author':'Ivaan Bayross',
        'Publisher': 'Pearson',
        'Year': 2010}
}
```

Python Dictionary Methods

Method	Description
clear()	Removes all the elements from the dictionary
copy()	Returns a copy of the dictionary
fromkeys()	Returns a dictionary with the specified keys and value
get()	Returns the value of the specified key
items()	Returns a list containing a tuple for each key value pair
keys()	Returns a list containing the dictionary's keys
pop()	Removes the element with the specified key
popitem()	Removes the last inserted key-value pair
setdefault()	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value
update()	Updates the dictionary with the specified key-value pairs
values()	Returns a list of all the values in the dictionary

Python Dictionary Methods

Sorting a Dictionary in Python

sorted() function

- The sorted() function can accept three parameters: the iterable, the key, and reverse. **sorted(iterable, key)**

```
sales = {'apple':10, 'banana':12, 'orange':6, 'grapes':100}
print(sorted(sales))

print(sorted(sales,key=sales.get))
```


Python Dictionary Methods

Dictionaries and lists share the following characteristics:

- Both are mutable.
- Both are dynamic. They can grow and shrink as needed.
- Both can be nested. A list can contain another list. A dictionary can contain another dictionary. A dictionary can also contain a list, and vice versa.

Dictionaries differ from lists primarily in how elements are accessed:

- List elements are accessed by their position in the list, via indexing.
- Dictionary elements are accessed via keys.

Python Sets

- Sets are used to store multiple items in a single variable.
- Set is one of 4 built-in data types in Python used to store collections of data, the other 3 are List, Tuple, and Dictionary, all with different qualities and usage.
- A set is a collection which is both *unordered* and *unindexed*.
- The items in a set do not have a defined order. Set items can appear in a different order every time you use them, and cannot be referred to by index or key.
- Every set element is unique (no duplicates) and must be immutable (cannot be changed). However, a set itself is mutable. We can add or remove items from it.
- Sets can also be used to perform mathematical set operations like union, intersection, symmetric difference, etc.
- The major advantage of using a set, as opposed to a list, is that it has a highly optimized method for checking whether a specific element is contained in the set. This is based on a data structure known as a hash table.

Python Sets continued …

Creating Python Sets

A set is created by placing all the items (elements) inside curly braces { }, separated by comma, or by using the built-in set() function.

It can have any number of items and they may be of different types (integer, float, tuple, string etc.). But a set cannot have mutable elements like lists, sets or dictionaries as its elements.

```
my_set={1,2,3}
print(my_set)

my_set={1.0, “Hello”, (1,2,3)}
 print(my_set)
```


Python Sets continued …

Creating an empty set is a bit tricky. Empty curly braces { } will make an empty dictionary in Python. To make a set without any elements, we use the set() function without any argument.

```
my_set=set()
```

Modifying a set in Python

- Sets are mutable. However, since they are unordered, indexing has no meaning.
- We cannot access or change an element of a set using indexing or slicing. Set data type does not support it.
- We can add a single element using the add() method, and multiple elements using the update() method. The update() method can take tuples, lists, strings or other sets as its argument. In all cases, duplicates are avoided.

Python Sets continued …

Modifying a set in Python

```
my_set={1,3}
print(my_set)

my_set.add(2)
print(my_set)

my_set.update([2,3,4,5,6])
print(my_set)

my_set.update([2,3],(7,8,9))
print(my_set)
```


Python Sets continued …

Removing elements from a set

- A particular item can be removed from a set using the methods ***discard()*** and ***remove()*** method
- The only difference between the two is that the ***discard()*** function leaves a set unchanged if the element is not present in the set. On the other hand, the ***remove()*** function will raise an error in such a condition (if element is not present in the set).

```
my_set={1,2,3,5,6}
print(my_set)

my_set.discard(4)
print(my_set)
my_set.remove(4)
```


Python Sets continued …

Python Set Operations

- Sets can be used to carry out mathematical set operations like union, intersection, difference and symmetric difference. We can do this with operators or methods.
- Let us consider the following two sets for the following operations.

```
A={1,2,3,4,5}
B={4,5,6,7,8}
```

Set Union – print(A | B)
Set Intersection - print(A & B)
Set Difference - print(A - B)
Set Symmetric Difference - print(A ^ B)
Set Membership Test – my_set=set("apple")

```
print('a' in my_set)
```

Python Sets continued …

Set Membership Test

```
my_set=set("apple")
print('a' in my_set)
print('A' in my_set)
print('T' not in my_set)
```

Iterating Through a Set

```
for letter in my_set:
    print(letter)

for n in A:
    print(n)
```


File Handling in Python

- Files are named locations on disk to store related information. They are used to permanently store data in a non-volatile memory (e.g. hard disk).
- Python too supports file handling and allows users to handle files i.e., to read and write files, along with many other file handling options, to operate on files.
- Python treats file differently as text or binary and this is important.
- Each line of code includes a sequence of characters and they form text file. Each line of a file is terminated with a special character, called the EOL or End of Line characters like comma {,} or newline character. It ends the current line and tells the interpreter a new one has begun.

File Handling in Python

In Python, a file operation takes place in the following order:

- Open a file
- Read or write (perform operation)
- Close the file

We use **open ()** function in Python to open a file in read or write mode. open ( ) will return a file object.

open() function accepts two arguments, file name and the mode, whether to read or write. There are three kinds of mode, that Python provides and how files can be opened:

- " **r** ", for reading.
- " **w** ", for writing.
- " **a** ", for appending.
- " **r+** ", for both reading and writing

File Handling in Python

We must keep in mind that the mode argument is not mandatory. If not passed, then Python will assume it to be " **r** " by default.

```
f= open("test.txt")              # opens file in current directory
f= open("C:/python37/test.txt")  # specifying full path
```

- The default is reading in text mode. In this mode, we get strings when reading from the file.
- On the other hand, binary mode ***"b"*** returns bytes and this is the mode to be used when dealing with non-text files like images or executable files.

```
f= open("img.bmp",'r+b')         # read and write in binary mode
```


File Handling in Python

Working of read() mode

```
f= open("D:/Python Programming/Scripts/test.txt", mode='r')
print(f.read())
print(f.read(5))
```

Creating a file using write() mode

```
f= open("D:/Python Programming/Scripts/test.txt", mode='w')
f.write("Testing write operation\n")
f.write("This command will add this line in file")
f.close()
```

- The close() command terminates all the resources in use and frees the system of this particular program.

File Handling in Python

Working of append() mode

```
f= open("D:/Python Programming/Scripts/test.txt", mode='a')
f.write("This command will add this line in file")
```

Using read along with with() function

```
with open("D:/Python Programming/Scripts/test.txt") as file:
    data= file.read()
print(data)
type(data)
```

File Handling in Python

split() using file handling

We can also split lines using file handling in Python. This splits the variable when space is encountered. You can also split using any characters as we wish.

```
with open("D:/Python Programming/Scripts/test.txt") as file:
    data= file.readlines()
    for line in data:
        word=line.split()
        print(word)
```

Deleting a file

```
import os
os.remove("D:/Python Programming/Scripts/test.txt")
```

UNIT-III

Learning Objectives

In this unit, we'll cover the following:

- **Object Oriented Programming Concepts**
- **Classes and Objects**
- **Inheritance**
- **Polymorphism**
- **Abstract Classes**
- **Threads**
- **Exception Handling**

Object Oriented Programming Concepts

- **Object-oriented programming** (OOP) is a method of structuring a program by bundling related properties and behaviors into individual **objects**.
- An object could represent a person with **properties** like a name, age, and address and **behaviors** such as walking, talking, breathing, and running. Or it could represent an email with properties like a recipient list, subject, and body and behaviors like adding attachments and sending.
- Object-oriented programming is an approach for modeling concrete, real-world things, like cars, as well as relations between things, like companies and employees, students and teachers, and so on. OOP models real-world entities as software objects that have some data associated with them and can perform certain functions.

OOPS in Python

- Python has been an object-oriented language since it existed. Because of this, creating and using classes and objects are very easy.
- Unlike procedure oriented programming, where the main emphasis is on functions, object oriented programming stresses on objects.
- The key takeaway is that objects are at the center of object-oriented programming in Python, not only representing the data, as in procedural programming, but in the overall structure of the program as well.

Overview of OOP Terminology

- **Class** − A user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.
- **Class variable** − A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods. Class variables are not used as frequently as instance variables are.
- **Data member** − A class variable or instance variable that holds data associated with a class and its objects.
- **Function overloading** − The assignment of more than one behavior to a particular function. The operation performed varies by the types of objects or arguments involved.

Overview of OOP Terminology continued…

- **Instance variable** − A variable that is defined inside a method and belongs only to the current instance of a class.
- **Inheritance** − The transfer of the characteristics of a class to other classes that are derived from it.
- **Instance** − An individual object of a certain class. An object obj that belongs to a class Circle, for example, is an instance of the class Circle.
- **Instantiation** − The creation of an instance of a class.
- **Method** − A special kind of function that is defined in a class definition.
- **Object** − A unique instance of a data structure that's defined by its class. An object comprises both data members (class variables and instance variables) and methods.
- **Operator overloading** − The assignment of more than one function to a particular operator.

Creating a Class in Python

Problem with primitive and advanced data types:
Aarti=[“Aarti Sharma”,24, “Officer”, 2018]

Supreet=[“Supreet Kaur”, “Sr. Manager”, 2010]

- It can make larger code files more difficult to manage.
- It can introduce errors if not every employee has the same number of elements in the list.

A great way to make this type of code more manageable and more maintainable is to use **classes**.

Classes

- Classes are used to create user-defined data structures. Classes define functions called **methods**, which identify the behaviors and actions that an object created from the class can perform with its data.
- A class is a blueprint for how something should be defined. It doesn't actually contain any data.
- While the class is the blueprint, an **instance** is an object that is built from a class and contains real data. Put another way, a class is like a form or questionnaire. An instance is like a form that has been filled out with information.

Some points on Python class:

- Classes are created by keyword class.
- Attributes are the variables that belong to a class.
- Attributes are always public and can be accessed using the dot (.) operator. Eg.: Myclass.Myattribute
-

Defining a Class

All class definitions start with the ***class*** keyword, which is followed by the name of the class and a colon. Any code that is indented below the class definition is considered part of the class.

The first string inside the class is called docstring and has a brief description about the class. Although not mandatory, this is highly recommended.

```
Class My_Class:
        ‘ I am creating my first class in Python’
        x=5
```

A class creates a new local namespace where all its attributes are defined. Attributes may be data or functions.

Creating an Object in Python

An Object is an instance of a Class. A class is like a blueprint while an instance is a copy of the class with *actual values*.

An object consists of :

- **State:** It is represented by the attributes of an object. It also reflects the properties of an object.
- **Behavior:** It is represented by the methods of an object. It also reflects the response of an object to other objects.
- **Identity:** It gives a unique name to an object and enables one object to interact with other objects.

Creating an Object in Python

Declaring Objects (Also called instantiating a class)

When an object of a class is created, the class is said to be instantiated. All the instances share the attributes and the behavior of the class. But the values of those attributes, i.e. the state are unique for each object. A single class may have any number of instances.

Create an object named p1, and print the value of x:

```
p1 = My_Class()
print(p1.x)
```

The class object could be used to access different attributes. Attributes may be data or method. Methods of an object are corresponding functions of that class.

The __init__() Function

Class functions that begin with double underscore _ are called special functions as they have special meaning.

Of one particular interest is the _init_() function. This special function gets called whenever a new object of that class is instantiated.

This type of function is also called constructors in Object Oriented Programming (OOP) languages like C++ and Java. We normally use it to initialize all the variables or object state.

Like methods, a constructor also contains a collection of statements(i.e. instructions) that are executed at the time of Object creation. It is run as soon as an object of a class is instantiated.

The __init__() Function

```
class Complex_Number:
        def _ini_(self, r=0,i=0):
                self.real = r
                self.imag=i

        def get_data(self):
                print(f'{self.real} + {self.imag}j')
num1=Complex_Number(2,3)
num1.get_data()

num2=Complex_Number(5)
num2.attr = 10
print(num2.real, num2.imag, num2.attr)
print(num1.attr)
```


Object Methods

Objects can also contain methods. Methods in objects are functions that belong to the object.

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

  def myfunc(self):
    print("Hello my name is " + self.name)

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


Self Parameter

The self parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class.

- It does not have to be named ***self***, you can call it whatever you like, but it has to be the first parameter of any function in the class
- Class methods must have an extra first parameter in the method definition. We do not give a value for this parameter when we call the method, Python provides it.
- If we have a method that takes no arguments, then we still have to have one argument.
- This is similar to this pointer in C++ and this reference in Java.
- When we call a method of this object as myobject.method(arg1, arg2), this is automatically converted by Python into MyClass.method(myobject, arg1, arg2) – this is all the special self is about.

Deleting Attributes and Objects

Any attribute of an object can be deleted anytime, using the ***del*** statement.

```
p1.age = 40
del p1.age
del p1.myfunc()
del p1
```

del() method

A class can implement the special method *__del__()*, called a destructor, that is invoked when the instance is about to be destroyed. destroyed

Deleting Attributes and Objects

```
class Point:
    def __init__(self,x=0,y=0)
        self.x=x
        self.y=y
    def __del__(self)
        class_name = self.__class__.__name__
        print class_name, "destroyed"
pt1=point()
pt2 = pt1
pt3 = pt1
print  id(pt1), id(pt2), id(pt3)
del pt1   del pt2   del pt3
```

This __del__() destructor prints the class name of an instance that is about to be

Built-In Class Attributes

- Every Python class keeps following built-in attributes and they can be accessed using dot operator like any other attribute −

- **__dict__** − Dictionary containing the class's namespace.
- **__doc__** − Class documentation string or none, if undefined.
- **__name__** − Class name.
- **__module__** − Module name in which the class is defined. This attribute is "__main__" in interactive mode.
- **__bases__** − A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.

Built-In Class Attributes

```
class Employee:
        'Common base class for all employees'
        empCount=0

        del __init__(self, name, salary):
                self.name=name
                self.salary.salary
                Employee.emCount +=1

        def displayCount(self):
                print ("Total employee = ", Employee.empCount)

        def displayEmployee(self):
                print ("name =" , self.name, ", Salary = ", self.salary)
```


Built-In Class Attributes

```
print "Employee.__doc__:" , Employee.__doc__
print "Employee.__name__:" , Employee.__name__
print "Employee.__module__:" , Employee.__module__
print "Employee.__bases__:" , Employee.__bases__
print "Employee.__dict__:" , Employee.__dict__
```


Access Modifiers in Python : Public, Private and Protected

- Python uses '_' symbol to determine the access control for a specific data member or a member function of a class. Access specifiers in Python have an important role to play in securing data from unauthorized access and in preventing it from being exploited.
- A Class in Python has three types of access modifiers –
  1. Public Access Modifier
  2. Protected Access Modifier
  3. Private Access Modifier

Public Access Modifier

The members of a class that are declared public are easily accessible from any part of the program. All data members and member functions of a class are public by default.

In previous examples :

self.name

self.age

myfunc()

Are all public members.

Protected Access Modifier

The members of a class that are declared protected are only accessible to a class derived from it. Data members of a class are declared protected by adding a single underscore '_' symbol before the data member of that class.

```
class student:
        _name = None
        _roll = None
        _branch = None
def __init__(self, name , roll, branch):
        self._name = name
        self._roll = roll
        self._branch = branch
def _displayRollAndBranch(self):
        print(self._roll)
        print(self._branch)
```


Protected Access Modifier

```
Class Derived_Stud(student):
        def __init__(self, name, roll, branch):
                student.__init__(self, name, roll, branch)

        def displayDetails:
                print("Name:", self.name)
                self._displayRollAndBranch()

Obj = Derived_Stud("Anuj", 170334, "Computer Applications")

Obj.displayDetails()
```

In the above program,_name, _roll and _branch are protected data members and _displayRollAndBranch() method is a protected method of the super class **Student**. The displayDetails() method is a public member function of the class **Derived_Stud.**

Private Access Modifier

The members of a class that are declared private are accessible within the class only, private access modifier is the most secure access modifier. Data members of a class are declared private by adding a double underscore '__' symbol before the data member of that class.

```
class student:
        __name = None
        __roll = None
        __branch = None
def __init__(self, name , roll, branch):
        self.__name = name
        self.__roll = roll
        self.__branch = branch
def _displayDetails(self):
        print(self.__name)
        print(self.__roll)
        print(self.__branch)
```


Private Access Modifier

```
def accessPrivateMethod(self)
        self.__discplayDetails()

Obj = student("Anuj", 170334, "Computer Applications")

Obj.accessPrivateMethod()
```

In the above program,__name, __roll and __branch are private data members and _displayDetails() method is a private member function of the class **student**. acessPrivateMethod() method is a public member function of the class **student** which can be accessed from anywhere within the program. The acessPrivateMethod() method accesses the private members of the class **student**.

Decorators in Python

- A **decorator** is a design pattern in **Python** that allows a user to add new functionality to an existing object without modifying its structure. Python has an interesting feature called **decorators** to add functionality to an existing code.
- This is also called **metaprogramming** because a part of the program tries to modify another part of the program at compile time.
- Decorators are very powerful and useful tool in Python since it allows programmers to modify the behavior of function or class. Decorators allow us to wrap another function in order to extend the behavior of the wrapped function, without permanently modifying it.
- Functions in Python can be used or passed as arguments.
  - A function is an instance of the Object type.
  - You can store the function in a variable.
  - You can pass the function as a parameter to another function.
  - You can return the function from a function.

Decorators in Python

Example 1: Treating the functions as objects.

```
def shout(text):
    return text.upper()
print(shout("Hello"))

yell=shout
print(yell("Hello"))
```

Example 2: Passing the function as argument

```
def shout(text):
    return text.upper()
def whisper(text):
    return text.lower()
def greet(func):
    greeting = func("Hi ! I am created by a function passed as an argument")

greet(shout)
greet(whisper)
```


Decorators in Python

Another example:

```
def inc(x):
    return x+1

def dec(x):
    return x-1

def operator(func,x):
    result=func(x)
    return result

>>>operator(inc,21)
>>>operator(dec,21)
```


Decorators in Python

Example 3: Returning functions from another functions.

```
def is_called():
    def is_returned():
            print("Hello")
    return is_returned

new = is_called()

new()
```

Here is_retured() is a nested function which is defined and returned each time we call is_called().

Decorators in Python

Functions and methods are called **callable** as they can be called.
Basically, a decorator takes in a function, adds some functionality and returns it.

```
def make_pretty(func):
    def inner():
        print("I got decorated")
        func()
    return inner
def ordinary()
    print("I am ordinary")

>>>ordinary()
>>>pretty=make_pretty(ordinary)
>>>pretty()
```

The function ordinary() got decora.ted and the returned function was given the name pretty().
We can see that the decorator function added some new functionality to the original function.
The decorator acts as a wrapper.

Decorators in Python

Generally, we decorate a function and reassign it as,

```
ordinary = make_pretty(ordinary)
```

We can use the @symbol along with the name of the decorator function and place it above the definition of the function to be decorated. For example,

```
@make_pretty
def ordinary()
    print("I am ordinary")
```

It is equivalent to following:

```
def ordinary()
    print("I am ordinary")
ordinary = make_pretty(ordinary)
```


Decorators with parameters

```
def smart_divide(func):
    def inner(a,b):
        print("I am going to divide ", a, " and ", b)
        if b==0:
            print("Can not divide")
            return
        return func(a,b)
    return inner

@smart_divide
def divide(a,b):
    print(a/b)
>>>divide(2,5)
>>>divide(2,0)
```

Different Types of Methods

There can be three types of methods in a class:

1. Instance methods
2. Class Methods
3. Static Methods

- The method **MyFunc()** is a regular instance method. That's the basic, no-frills method type you'll use most of the time. You can see the method takes one parameter, self, which points to an instance of class **Person** when the method is called (but of course instance methods can accept more than just one parameter).
- Through the self parameter, nstance methods can freely access attributes and other methods on the same object. This gives them a lot of power when it comes to modifying an object's state.
- Not only can they modify object state, instance methods can also access the class itself through the self.__class__ ttribute. This means instance methods can also modify class state.

Class Methods

```
@classmethod
Def classmethod(cls):
        return cls
```

Above method is marked with a **@classmethod** decorator to flag it as a *class method*.

Instead of accepting a **self** parameter, class methods take a **cls** parameter that points to the class—and not the object instance—when the method is called.

Because the class method only has access to this **cls** argument, it can't modify object instance state. That would require access to self. However, class methods can still modify class state that applies across all instances of the class.

Static Methods

```
@staticmethod
Def staticmethod():
        return
```

Above method is marked with a **@staticmethod** decorator to flag it as a *static method*.

This type of method takes neither a **self** or a **cls** parameter (but of course it's free to accept an arbitrary number of other parameters).

Therefore a static method can neither modify object state nor class state. Static methods are restricted in what data they can access - and they're primarily a way to namespace your methods.

Class Vs Static Methods

- A class method takes cls as first parameter while a static method needs no specific parameters.
- A class method can access or modify class state while a static method can't access or modify it.
- In general, static methods know nothing about class state. They are utility type methods that take some parameters and work upon those parameters. On the other hand class methods must have class as parameter.
- We use @classmethod decorator in python to create a class method and we use @staticmethod decorator to create a static method in python.

Class Vs Static Methods

When to use what?

- We generally use class method to create factory methods. Factory methods return class object (similar to a constructor) for different use cases.
- We generally use static methods to create utility functions.

How to define a class method and a static method?

To define a class method in python, we use @classmethod decorator and to define a static method we use @staticmethod decorator.

Class Vs Static Methods

```
from datetime import date
class person:
        def __init__(self, name, age):
                self.name=name
                self.age=age
        @classmethod
        def fromBirthYear(cls, name, year):
                return cls(name, date.today().year)

        @staticmethod
        def isAdult(age):
                return age>=18
person1= person("Anuj", 21)
person2= person.fromBirthYear(("Anuj", 2000)
print(person1.age)
print(person2.age)
```

Data Hiding in Python

Nothing in Python is truly private; internally, the names of private methods and attributes are mangled and unmangled on the fly to make them seem inaccessible by their given names.

In Python, we use double underscore (Or __) before the attributes name and those attributes will not be directly visible outside.

```
class myClass:
        __hiddenVariable = 0
        def add(self, increment):
                self.__hiddenVariable += increment
                print(self.__hiddenVariable)
myObject = myClass()
myObject.add(2)
myObject.add(5)
print(myObject.__hiddenVariable)
```


Data Hiding in Python

In the previous code, we tried to access hidden variable outside the class using object and it threw an exception.

Now try,
Print(myObject._myClass__hiddenVariable)

Python protects those members by internally changing the name to include the class name. You can access such attributes as object._className__attrName.

Inheritance in Python

Inheritance is the capability of one class to derive or inherit the properties from another class. The benefits of inheritance are:
It represents real-world relationships well.

- It provides **reusability** of a code. We don't have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.
- It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.

In inheritance, the child class acquires the properties and can access all the data members and functions defined in the parent class. A child class can also provide its specific implementation to the functions of the parent class.

Different forms of Inheritance:

1. **Single inheritance**: When a child class inherits from only one parent class, it is called single inheritance.

```
class derived-class(base class):
        <class-suite>
```

Inheritance in Python

2. Multiple inheritance: When a child class inherits from multiple parent classes, it is called multiple inheritance. Unlike Java and like C++, Python supports multiple inheritance. We specify all parent classes as a comma-separated list in the bracket.

```
class derive-class(<base class 1>, <base class 2>, ..... <base class n>):
        <class - suite>
```

3. Multilevel inheritance: Multi-level inheritance is archived when a derived class inherits another derived class. There is no limit on the number of levels up to which, the multi-level inheritance is archived in python.

```
class class1:
  <class-suite>
class class2(class1):
  <class suite>
class class3(class2):
  <class suite>
```


Inheritance in Python continued …

A simple example of inheritance in Python:

```
class person:
        def __init__(self, fname, lname):
                self.fname=fname
                self.lname=lname
        def printName(self):
                print(self.fname, self.lname)

x = person('Anuj', 'Rastogi')
x.printName()

class student(person):
        pass
y=student('Aarti','Sharma')
y.printName()
```


Inheritance in Python continued …

So far we have created a child class that inherits the properties and methods from its parent. Now, we want to add the __init__() function to the child class

```
class student(person):
        def __init__(self, lname,fname):
```

When you add the __init__() function to the child class, it will no longer inherit the parent's __init__() function. The child's __init__() function **overrides** the parent's __init__() function.

To keep the inheritance of the parent's __init__(), add a call to the parent's __init__()

```
class student(person):
        def __init__(self, lname,fname):
                person. __init__(self, lname,fname)
```

Inheritance in Python continued …

Python also has a ***super()*** function that will make the child class inherit all the methods and properties from its parent:

```
class student(person):
    def __init__(self, lname, fname):
        super(). __init__(lname, fname)
```

By using the ***super()*** function , you do not have to use the name of the parent element, it will automatically inherit the methods and properties from its parent.

Add Properties

```
class student(person):
    def __init__(self, lname, fname):
        super(). __init__(lname, fname)
        self.graduationyear = 2022
```

graduationyear should be passed to __init__() function

Inheritance in Python continued …

Add Methods

```
class student(person):
    def __init__(self, lname, fname):
        super(). __init__(lname, fname)
        self.graduationyear = 2022
    def welcome(self):
        print("Welcome", self.firstname, self.lastname, "to the class of", self.graduationyear)
```

If you add a method in the child class with the same name as a function in the parent class, the inheritance of the parent method will be overridden.

Two built-in functions ***isinstance()*** and ***issubclass()*** are used to check inheritances.

Inheritance in Python continued …

Method Overriding

- We can provide some specific implementation of the parent class method in our child class.
- When the parent class method is defined in the child class with some specific implementation, then the concept is called method overriding.
- We may need to perform method overriding in the scenario where the different definition of a parent class method is needed in the child class.

Inheritance in Python continued …

```
class Bank:
    def getroi(self):
        return 10;
class SBI(Bank):
    def getroi(self):
        return 7;

class ICICI(Bank):
    def getroi(self):
        return 8;
b1 = Bank()
b2 = SBI()
b3 = ICICI()
print("Bank Rate of interest:",b1.getroi());
print("SBI Rate of interest:",b2.getroi());
print("ICICI Rate of interest:",b3.getroi());
```


Polymorphism in Python

The word polymorphism means having many forms. In programming, polymorphism means same function name (but different signatures) being used for different types.

Polymorphism is a very important concept in programming. It refers to the use of a single type entity (method, operator or object) to represent different types in different scenarios.

Polymorphism and Inheritance

- The child classes in Python also inherit methods and attributes from the parent class. We can redefine certain methods and attributes specifically to fit the child class, which is known as **Method Overriding**.
- Polymorphism allows us to access these overridden methods and attributes that have the same name as the parent class.

Polymorphism in Python continued…

Polymorphism with Class Methods

```
class India:
        def capital(self):
                print('New Delhi')
        def language(self):
                print('Hindi, English and other regional languages')
 class USA:
        def capital(self):
                print('Washington, D.C.')
        def language(self):
                print('English')
obj_ind=India()
obj_usa=USA()
for country in (obj_ind, obj_usa):
        country.capital()
        country.language()
```

Polymorphism in Python continued…

Built-in polymorphic functions:

```
print(len('BVICAM'))
print(len(['Python', 'Java', 'C++']))
print(len('Monday','Tuesday','Wednesday','Thursday','Friday','Saturday','Sunday'))
```

User defined polymorphic functions :

```
def add(x,y,z=0):
        return x+y+z
print(add(2,3))
print(add(2,3,4))
```


Operator Overloading

- Operator Overloading means giving extended meaning beyond their predefined operational meaning. For example operator + is used to add two integers as well as join two strings and merge two lists. It is achievable because '+' operator is overloaded by int class and str class.
- The same built-in operator or function shows different behavior for objects of different classes, this is called *Operator Overloading*.

```
class point:
        def __init__(self, x,y):
                self.x=x
                self.y=y
p1=point(2,3)
p2=point(5,8)

print(p1+p2)
```


- We can change the meaning of an operator in Python depending upon the operands used.
- When we use an operator on user defined data types then automatically a special function or magic function associated with that operator is invoked. Changing the behavior of operator is as simple as changing the behavior of method or function.
- We define methods in your class and operators work according to that behavior defined in methods. When we use + operator, **the magic method __add__** is automatically invoked in which the operation for + operator is defined. There by changing this magic method's code, we can give extra meaning to the + operator.

```
        def __add__(self, other):
                x = self.x + other.x
                y = self.y + other.y
                return point(x,y)
```

Now try

```
print(p1+p2)
```

Magic Functions for Operator Overloading

Binary Operators:

+	__add__(self, other)
–	__sub__(self, other)
*	__mul__(self, other)
/	__truediv__(self, other)
//	__floordiv__(self, other)
%	__mod__(self, other)
**	__pow__(self, other)
>>	__rshift__(self, other)
<<	__lshift__(self, other)
&	__and__(self, other)
|	__or__(self, other)
^	__xor__(self, other)

Magic Functions for Operator Overloading

Comparison Operators :

<	__LT__(SELF, OTHER)
>	__GT__(SELF, OTHER)
<=	__LE__(SELF, OTHER)
>=	__GE__(SELF, OTHER)
==	__EQ__(SELF, OTHER)
!=	__NE__(SELF, OTHER)

Unary Operators :

–	__NEG__(SELF, OTHER)
+	__POS__(SELF, OTHER)
~	__INVERT__(SELF, OTHER)

Magic Functions for Operator Overloading

Assignment Operators :

-=	__ISUB__(SELF, OTHER)
+=	__IADD__(SELF, OTHER)
*=	__IMUL__(SELF, OTHER)
/=	__IDIV__(SELF, OTHER)
//=	__IFLOORDIV__(SELF, OTHER)
%=	__IMOD__(SELF, OTHER)
**=	__IPOW__(SELF, OTHER)
>>=	__IRSHIFT__(SELF, OTHER)
<<=	__ILSHIFT__(SELF, OTHER)
&=	__IAND__(SELF, OTHER)
|=	__IOR__(SELF, OTHER)
^=	__IXOR__(SELF, OTHER)

Abstract Classes

- **A class is called an Abstract class if it contains one or more abstract methods**. An abstract method is a method that is declared, but contains no implementation. Abstract classes may not be instantiated, and its abstract methods must be implemented by its subclasses.
- An abstract class can be considered as a blueprint for other classes.
- While we are designing large functional units we use an abstract class. When we want to provide a common interface for different implementations of a component, we use an abstract class.

How Abstract Base classes work

By default, Python does not provide abstract classes. Python comes with a module that provides the base for defining Abstract Base classes(ABC) and that module name is abc. ***abc*** works by decorating methods of the base class as abstract and then registering concrete classes as implementations of the abstract base. A method becomes abstract when decorated with the keyword @abstractmethod. For Example –

```
Import abc
Class polygon(abc.ABC)
    @abc.abstractmethod
    def noofsides(self):
        pass
```

You may also provide class methods and static methods in abstract base class by decorators @abstractclassmethod and @abstractstatic method decorators respectively.

Abstact Class continued…

```
class Triangle(polygon):
    def noofsides(self):
        print("I have 3 sides")

class Pentagon(polygon):
    def noofsides(self):
        print("I have 5 sides")

class Hexagon(polygon):
    def noofsides(self):
        print("I have 6 sides")

R = Triangle()
R.noofsides()
```

Threads in Python

Thread

In computing, a **process** is an instance of a computer program that is being executed. Any process has 3 basic components:

- An executable program.
- The associated data needed by the program (variables, work space, buffers, etc.)
- The execution context of the program (State of process)

A **thread** is an entity within a process that can be scheduled for execution. Also, it is the smallest unit of processing that can be performed in an OS (Operating System).

In simple words, a **thread** is a sequence of such instructions within a program that can be executed independently of other code. For simplicity, you can assume that a thread is simply a subset of a process!

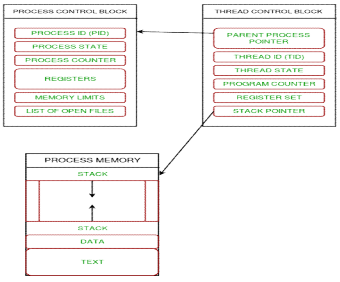
Threads continued…

A thread contains all this information in a **Thread Control Block (TCB)**:

- **Thread Identifier:** Unique id (TID) is assigned to every new thread
- **Stack pointer:** Points to thread's stack in the process. Stack contains the local variables under thread's scope.
- **Program counter:** a register which stores the address of the instruction currently being executed by thread.
- **Thread state:** can be running, ready, waiting, start or done.
- **Thread's register set:** registers assigned to thread for computations.
- **Parent process Pointer:** A pointer to the Process control block (PCB) of the process that the thread lives on.

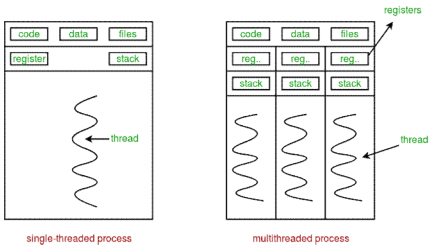
Threads continued…

Consider the diagram below to understand the relation between process and its thread:



Multithreading

- Multiple threads can exist within one process where:
- Each thread contains its own **register set** and **local variables (stored in stack)**.
- All thread of a process share **global variables (stored in heap)** and the **program code**.



Multithreading continued…

Multithreading is defined as the ability of a processor to execute multiple threads concurrently.

Running several threads is similar to running several different programs concurrently, but with the following benefits −

- Multiple threads within a process share the same data space with the main thread and can therefore share information or communicate with each other more easily than if they were separate processes.
- Threads sometimes called light-weight processes and they do not require much memory overhead; they are cheaper than processes.

A thread has a beginning, an execution sequence, and a conclusion. It has an instruction pointer that keeps track of where within its context it is currently running.

- It can be pre-empted (interrupted)
- It can temporarily be put on hold (also known as sleeping) while other threads are running - this is called yielding.

Multithreading continued…

There are two ways of accessing Python threads. These are by using:

- Thread Module
  t = thread.strat_new_thread(function, args [, kwargs])
- Threading module
  t = threading.Thread(function, args [, kwargs])

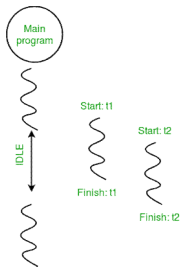
It is to be noted that the 'tread' module has been considered as of lesser use, and hence users get to use the 'threading' module instead. Another thing has to keep in mind that the module 'thread' treats the thread as a function, whereas the 'threading' is implemented as an object.

Multithreading continued…

```
import threading
def print_cube(num):
        print('Cube = ', num * num * num)
def print_square(num):
        print('Square = ', num * num )

If __name__= "__main__" :
        t1 = threading.Thread(target=print_square, args=(10,))
         t2 = threading.Thread(target=print_cube, args=(10,))

        t1.start()
        t2.start()

        t1.join()
        t2.join()
        print("Done!")
```


Multithreading continued…

Consider the diagram below for a better understanding of how this program works:



Another Example of Multithreading

```
import threading
import os
def task1():
        print("Task 1 assigned to thread: ", threading.current_thread().name)
        print("ID of process running Task 1", os.getpid())
def task2():
        print("Task 2 assigned to thread: ", threading.current_thread().name)
        print("ID of process running Task 2", os.getpid())
if __name__= "__main__" :
        print("ID of process running MAIN program: ", os.getpid())
        print("Main thread name : ", threading.current_thread().name)
        t1= threading.Thread(target=task1, name='t1')
        t2=threading.Thread(target=task2,name='t2')
        t1.start()
        t2.start()
        t1.join()
        t2.join()
```

Python Program for Various Thread States

- There are five thread states - **new, runnable, running, waiting and dead**.
- Among these five Of these five, we will majorly focus on three states - running, waiting and dead.
- A thread gets its resources in the running state, waits for the resources in the waiting state; the final release of the resource, if executing and acquired is in the dead state.
- The following Python program with the help of start(), sleep() and join() methods will show how a thread entered in running, waiting and dead state respectively.

Step 1 − Import the necessary modules, for example : <threading> and <time>

Step 2 − Define a function, which will be called while creating a thread.

Step 3 − We are using the sleep() method of time module to make our thread waiting for say 2 seconds.

Step 4 − Now, we are creating a thread (e.g T1), which takes the argument of the function defined above.

Step 5 − Now, with the help of the start() function we can start our thread. It will produce the message, which has been set by us while defining the function.

Step 6 − Now, at last we can kill the thread with the join() method after it finishes its execution.

Daemon Thread in Python

- A **daemon thread** is a thread that dies whenever the main thread dies, it is also called a non-blocking thread.
- Usually, the main thread should wait for other threads to finish in order to quit the program, but if you set the daemon flag, you can let the thread do its work and forget about it, and when the program quits, it will be killed automatically.
- For example, you may want to make a thread that watches for log files in your program, and alert you when a critical error is occurred.
- Usually our main program implicitly waits until all other threads have completed their work.
- The default setting for a thread is non-daemon. To designate a thread as a daemon, we call its **setDaemon()** method with a boolean argument.

Example: test-Daemon.py

Join() method in Python

On invoking the **join()** method, the calling thread gets blocked until the thread object (on which the thread is called) gets terminated. The thread objects can terminate under any one of the following conditions:

- Either normally.
- Through an ill-handled exception.
- Till the optional timeout occurs.

Hence the join() method indicates wait till the thread terminates. We can also specify a timeout value to the join() method. In such a situation the calling thread may ask the thread to stop by sending a signal through an event object. The join() method can be called multiple times.

threading.enumerate()

threading.enumerate() returns a list of all Thread objects currently alive. The list includes daemonic threads, and the main thread.
It excludes terminated threads and threads that have not yet been started.

```
for t in threading.enumerate():
        if t is main_thread:
            continue
        logging.debug('Name of the thread = %s', t.getName())
```

For example : test-Enumerate.py

Thread Synchronization in Python

Thread synchronization is defined as a mechanism which ensures that two or more concurrent threads do not simultaneously execute some particular program segment known as **critical section**.

Following issues may arise while implementing concurrent programming or applying synchronizing primitives:

1. Dead Lock
2. Race condition

In Python, we can implement synchronization by using the following concepts

- **Lock**
- **RLock**
- **Semaphore**

Synchronization By using Locks in python:

Locks are the most fundamental synchronization mechanism provided by the threading module. We can create Lock object as follows,

l=Lock()

The Lock object can be held by only one thread at a time. If any other thread wants the same lock then it will have to wait until the other one releases it. It's similar to waiting in line to book a train ticket, public telephone booth etc.

- **acquire() method:** A Thread can acquire the lock by using acquire() method
  l.acquire()
- **release() method:** A Thread can release the lock by using release() method.
  l.release()

Examples : Test-Lock.py and Test-Lock1.py

Problem with Simple Lock in Python:

- The standard lock object does not care which thread is currently holding that lock. If the lock is being held by one thread, and if any other thread tries to accquire the lock, then it will be blocked, even if it's the same thread that is already holding the lock.
- So, if the Thread calls recursive functions or nested access to resources, then the thread may try to acquire the same lock again and again, which may result in blocking of our thread. Hence Traditional Locking mechanism won't work for executing recursive functions.

Synchronization By using RLock concept in Python:

- To overcome the above problem of Simple Lock, we should go for RLock(Reentrant Lock). Reentrant means the thread can acquire the same lock again and again. This will block the thread only if the lock is held by any other thread. Reentrant facility is available only for owner thread but not for other threads.
- This RLock keeps track of recursion level and hence for every acquire() there should be a release() call available.
- The number of acquire() calls and release() calls should be matched then for the lock to be released i.e if there are two accquire calls then there should be two release calls for the lock to be released. If there is only one release call for two accquire calls then the lock wont be released.

Example : test-RIock()

Difference between Lock and RLock in Python:

Lock	RLock
A Lock object can be acquired by only one thread at a time. Even the owner thread also cannot acquire multiple times.	A RLock object can be acquired by only one thread at a time, but the owner thread can acquire the same lock object multiple times.
Not suitable to execute recursive functions and nested access calls	Best suitable to execute recursive functions and nested access calls
In this case the Lock object will only see whether its Locked or unlocked and it will never hold or take care of owner thread and recursion level.	In this case RLock object will see whether its Locked or unlocked and also about owner thread information and recursion level.

Synchronization by using Semaphore in Python:

- Semaphore provides threads with synchronized access to a limited number of resources.
- A semaphore is just a variable. The variable reflects the number of currently available resources. For example, a parking lot with a display of number of available slots on a specific level of a shopping mall is a semaphore.
- The value of semaphore cannot go less than zero and greater then the total number of the available resources.
- The semaphore is associated with two operations – **acquire** and **release**.
- When one of the resources synchronized by a semaphore is **"acquired"** by a thread, the value of the semaphore is decremented.
- When one of the resources synchronized by a semaphore is **"released"** by a thread the value of the semaphore is incremented.

Semaphores in Python:

- The Semaphore class of the Python threading module implements the concept of semaphore.
- It has a constructor and two methods acquire() and release().
- The acquire() method decreases the semaphore count if the count is greater than zero. Else it blocks till the count is greater than zero.
- The release() method increases the semaphore count and wakes up one of the threads waiting on the semaphore.

Way to create an object of Semaphore :

1. object_name.Semaphore()
   In this case, by default value of the count variable is 1 due to which only one thread is allowed to access. It is exactly the same as the **Lock** concept.
2. object_name.Semaphore(n)
   In this case, a Semaphore object can be accessed by n Threads at a time. The remaining Threads have to wait until releasing the semaphore.

Example : test-Semaphores.py

Bounded Semaphore in Python:

Try this :

```
from threading import *
s=Semaphore(2)
s.acquire()
s.acquire()
s.release()
s.release()
s.release()
s.release()
print("End")
```

In **Normal Semaphore**, discussed above, the release method can be called any number of times to increase the counter, irrespective of the acquire method. Sometimes, the number of release() calls can exceed

It is valid because in normal semaphore we can call release() any number of times. This may result in programming errors or may raise confusions. So, it is always recommended to use **Bounded Semaphore** which raises an error if the number of release() calls exceeds the number of acquire() calls. the number of acquire() calls also.

Bounded Semaphore in Python:

Now try this:

```
from threading import *
s=BoundedSemaphore(2)
s.acquire()
s.acquire()
s.release()
s.release()
s.release()
s.release()
print("End")
```

Example: test-semaphores2.py

The major idea of synchronization is to overcome data inconsistency problems. But the disadvantage of synchronization is it increases waiting time of threads and creates performance problems. Hence it is recommended to use synchronization only if the requirement demands.

Thread Synchronization using Event Object

The Event class object provides a simple mechanism which is used for communication between threads where one thread signals an event while the other threads wait for it.

So, when one thread which is intended to produce the signal produces it, then the waiting thread gets activated.

An internal flag is used by the event object known as the **event flag** which can be set as true using the set() method and it can be reset to false using the clear() method

The wait() method blocks a thread until the event flag for which it is waiting is set true by any other thread.

There are few useful functions used along with an event object:

1. isSet()
2. set()
3. clear()
4. wait([Timeout])

Thread Synchronization using Event Object

isSet Method:

This method returns true if and only if the internal flag is true.

set() Method:

When this method is called for any event object then the internal flag is set to true. And as soon as set() methos is called for any event all threads waiting for it are awakened.

clear() Method:

This method resets the internal flag to false. Subsequently, threads calling wait() on the event for which clear() is called, it will block until the internal flag is not true again.

wait([Timeout]) Method:

- When we have to make any thread wait for an event, we can do so by calling this method on that event which has the internal flag set to false, doing so blocks the thread until the internal flag is true for the event.
- If the internal flag is true on entry, then the thread will never get blocked. Otherwise, it is blocked until another thread calls set() to set the flag to true, or until the optional timeout occurs. The timeout argument specifies a timeout for the operation in seconds.

Example : test-event.py

Exception Handling

Error in Python can be of two types i.e. Syntax errors and Exceptions. Errors are the problems in a program due to which the program will stop the execution. On the other hand, exceptions are raised when some internal events occur which changes the normal flow of the program.

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

```
marks =1000
a=marks / 0
print(a)
```


Exception Handling

Python provides two very important features to handle any unexpected error in your Python programs and to add debugging capabilities in them −

- **Exception Handling**
- **Assertions**

Python provides a way to handle the exception so that the code can be executed without any interruption. If we do not handle the exception, the interpreter doesn't execute all the code that exists after the exception.

Python has many **built-in exceptions** that enable our program to run without interruption and give the output.

Exception Handling

Common Exceptions

Python provides the number of built-in exceptions, but here we are describing the common standard exceptions. A list of common exceptions that can be thrown from a standard Python program is given below.

- **ZeroDivisionError:** Occurs when a number is divided by zero.
- **NameError:** It occurs when a name is not found. It may be local or global.
- **IndentationError:** If incorrect indentation is given.
- **IOError:** It occurs when Input Output operation fails.
- **EOFError:** It occurs when the end of the file is reached, and yet operations are being performed.
- **ArithmeticError:** Base class for all errors that occur for numeric calculation.
- **OverflowError:** Raised when a calculation exceeds maximum limit for a numeric type.
- **IndexError:** Raised when an index is not found in a sequence.
- **SyntaxError**: Raised when there is an error in Python syntax.
- **ValueError:** Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.

Exception Handling

Handling an exception

- If you have some *suspicious* code that may raise an exception, you can defend your program by placing the suspicious code in a **try:** block. After the try: block, include an **except:** statement, followed by a block of code which handles the problem as elegantly as possible.

Syntax

Here is simple syntax of *try....except...else* blocks −

```
try:
 You do your operations here;
 ......................
except ExceptionI:
    If there is ExceptionI, then execute this block.
except ExceptionII:
    If there is ExceptionII, then execute this block.
     ......................
else:
    If there is no exception then execute this block.
```


Exception Handling

Here are few important points about the syntax −

- A single try statement can have multiple except statements. This is useful when the try block contains statements that may throw different types of exceptions.
- You can also provide a generic except clause, which handles any exception.
- After the except clause(s), you can include an else-clause. The code in the else-block executes if the code in the try: block does not raise an exception.
- The else-block is a good place for code that does not need the try: block's protection.

```
try:
  fh = open("testfile", "w")
  fh.write("This is my test file for exception handling!!")
except IOError:
  print "Error: can\'t find file or read data"
else:
  print "Written content in the file successfully"
  fh.close()
```


The *except* Clause with No Exceptions

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

```
try:
   You do your operations here;
   ......................
except:
   If there is any exception, then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```

This kind of a **try-except** statement catches all the exceptions that occur. Using this kind of try-except statement is not considered a good programming practice though, because it catches all exceptions but does not make the programmer identify the root cause of the problem that may occur.

The *except* Clause with Multiple Exceptions

```
try:
   You do your operations here;
   ......................
Except(<Exception 1>, <Exception 2>, <Exception 3>):
   Block of statements
   ......................
else:
   If there is no exception then execute this block.
```


The try-finally Clause

Python provides the optional **finally** statement, which is used with the **try** statement. It is executed no matter what exception occurs and used to release the external resource. The finally block provides a guarantee of the execution.

```
try:
    fileptr = open("file2.txt","r")
    try:
        fileptr.write("Hi I am good")
    finally:
        fileptr.close()
        print("file closed")
except:
    print("Error")
```


Raising exceptions

- An exception can be raised forcefully by using the **raise** clause in Python. It is useful in in that scenario where we need to raise an exception to stop the execution of the program.
- For example, there is a program that requires 2GB memory for execution, and if the program tries to occupy 2GB of memory, then we can raise an exception to stop the execution of the program.
- The syntax to use the raise statement is given below.

```
Raise Exception_class, <value>
```

Points to remember

- To raise an exception, the raise statement is used. The exception class name follows it.
- An exception can be provided with a value that can be given in the parenthesis.
- To access the value "**as**" keyword is used. "**e**" is used as a reference variable which stores the value of the exception.
- We can pass the value to an exception to specify the exception type.

Raising exceptions

```
try:
    age = int(input("Enter the age:"))
    if(age<18):
        raise ValueError
    else:
        print("the age is valid")
except ValueError:
    print("The age is not valid")
```

Raise the exception with message

```
try:
    num = int(input("Enter a positive integer: "))
    if(num <= 0):
        raise ValueError("That is  a negative number!")
except ValueError as e:
    print(e)
```


Custom Exception

The Python allows us to create our exceptions that can be raised from the program and caught using the except clause. However, we suggest you read this section after visiting the Python object and classes.

```
class ErrorInCode(Exception):
    def __init__(self, data):
        self.data = data
    def __str__(self):
        return repr(self.data)

try:
    raise ErrorInCode(2000)
except ErrorInCode as ae:
    print("Received error:", ae.data)
```


Assertions in Python

- An assertion is a sanity-check that you can turn on or turn off when you are done with your testing of the program.
- The **assert** keyword is used when debugging code.
- The **assert** keyword lets you test if a condition in your code returns True, if not, the program will raise an AssertionError.
- Assertions are carried out by the assert statement, the newest keyword to Python, introduced in version 1.5.
- Programmers often place assertions at the start of a function to check for valid input, and after a function call to check for valid output.

The *assert* Statement

- When it encounters an assert statement, Python evaluates the accompanying expression, which is hopefully true. If the expression is false, Python raises an *AssertionError* exception.
- The **syntax** for assert is −

  assert Expression[, Argument]

Assertions in Python

Here is a function that converts a temperature from degrees Kelvin to degrees Fahrenheit. Since zero degrees Kelvin is as cold as it gets, the function bails out if it sees a negative temperature −

```
def KelvinToFahrenheit(Temperature):
  assert (Temperature >= 0),"Colder than absolute zero!"
  return ((Temperature-273)*1.8)+32

print KelvinToFahrenheit(273)
print int(KelvinToFahrenheit(505.78))
print KelvinToFahrenheit(-5)
```


Exception Class Hierarchy in Python:

- All exception classes are derived from the BaseException class.
- The BaseException is the base class of all other exceptions. User defined classes cannot be directly derived from this class, to derive user defied class, we need to use Exception class.

Python Exception Hierarchy.docx

Composition in Python

- **Composition** is an object oriented design concept that models a **has a** relationship. In composition, a class known as **composite** contains an object of another class known to as **component**. In other words, a composite class **has a** component of another class.
- Composition allows composite classes to reuse the implementation of the components it contains. The composite class doesn't inherit the component class interface, but it can leverage its implementation.
- The composition relation between two classes is considered loosely coupled. That means that changes to the component class rarely affect the composite class, and changes to the composite class never affect the component class.
- This provides better adaptability to change and allows applications to introduce new requirements without affecting existing code.
- When looking at two competing software designs, one based on inheritance and another based on composition, the composition solution usually is the most flexible.
- Example : test-Composition.py

Iterators in Python

A process that is repeated more than one time by applying the same logic is called an Iteration. In programming languages like python, a loop is created with few conditions to perform iteration till it exceeds the limit. If the loop is executed 6 times continuously, then we could say the particular block has iterated 6 times.

```
a = [0, 5, 10, 15, 20]
for i in a:
    if i % 2 == 0:
        print(str(i)+' is an Even Number')
    else:
        print(str(i)+' is an Odd Number')
```


Iterators in Python

- An iterator is an object which contains a countable number of values and it is used to iterate over iterable objects like list, tuples, sets, etc.
- Iterators are implemented using a class and a local variable for iterating is not required here, It follows lazy evaluation where the evaluation of the expression will be on hold and stored in the memory until the item is called specifically which helps us to avoid repeated evaluation.
- As lazy evaluation is implemented, it requires only 1 memory location to process the value and when we are using a large dataset then, wastage of RAM space will be reduced the need to load the entire dataset at the same time will not be there.

How to use an iterator-

- **iter()** keyword is used to create an iterator containing an iterable object.
- **next()** keyword is used to call the next element in the iterable object.
- After the iterable object is completed, to use them again reassign them to the same object.

```
iter_list = iter(['Java', 'Python', 'C++'])
print(next(iter_list))
print(next(iter_list))
```


Generators in Python

- It is another way of creating iterators in a simple way where it uses the keyword "yield" instead of returning it in a defined function.
- Generators are implemented using a function.
- Just as iterators, generators also follow lazy evaluation.
- The yield function returns the data without affecting or exiting the function. It will return a sequence of data in an iterable format where we need to iterate over the sequence to use the data as they won't store the entire sequence in the memory.

Example : test-Generator.py

UNIT-IV

Overview of the Unit

In this unit, we'll cover the following:

- **Advanced Python**
- **NumPy Library**
- **Pandas Library**
- **Data Visualization**
- **GUI Programming**

List Comprehensions in Python

List comprehension is an elegant way to define and create a list in python. We can create lists just like mathematical statements and in one line only. The syntax of list comprehension is easier to grasp. A list comprehension generally consists of these parts :

- Output expression,
- Input sequence,
- A variable representing a member of the input sequence and
- An optional predicate part.

General syntax is:

Lst = [expression(i) for i in another_list if filter(i)]

List Comprehension

Example -1:

Lst = [x**2 for x in range(1,11) if x% 2 == 1]

In the above example,

- x ** 2 is the expression.
- range (1, 11) is input sequence or another list.
- x is the variable.
- if x % 2 == 1 is predicate part.

This is the power of list comprehension. It can identify when it receives a string or a tuple and work on it like a list.

Nested IF with List Comprehension

```
num_list = [y for y in range(100) if y%2==0 if y%5==0]
print(num_list)
```


List Comprehension

Example -2 :

Suppose, we want to separate the letters of the word 'human' and add the letters as items of a list. The first thing that comes in mind would be using for loop.

```
h_letters = []
for letter in 'human':
        h.letters.append(letter)
print(h_letters)
```

Python has an easier way to solve this issue using List Comprehension.

```
h_letters = [letter for letter in 'human' ]
print(h_letters)
```


List Comprehension

In **example-2**, we can see that 'human' is a string, not a list. This is the power of list comprehension. It can identify when it receives a string or a tuple and work on it like a list.

You can do that using loops.

However, not every loop can be rewritten as list comprehension. But as you learn and get comfortable with list comprehensions, you will find yourself replacing more and more loops with this elegant syntax.

Advantages of List Comprehension

Advantages of List Comprehension

- More time efficient and space efficient than loops.
- Require fewer lines of code.
- Transforms iterative statement into a formula.

Key Points to Remember

- we should avoid writing very long list comprehensions in one line to ensure that code is user-friendly.
- Every list comprehension can be rewritten in for loop, but every for loop can't be rewritten in the form of list comprehension.

map() Function

The advantage of the lambda function can be seen when it is used in combination with the map() function. map() is a function which takes two arguments:

```
r = map(func, seq)
```

- The first argument func is the name of a function and the second a sequence (e.g. a list) seq. map() applies the function func to all the elements of the sequence seq.
- Before Python3, map() used to return a list, where each element of the result list was the result of the function func applied on the corresponding element of the list or tuple "seq". With Python 3, map() returns an iterator.

map() Function

- The map() function executes a specified function for each item in an iterable. The item is sent to the function as a parameter.

Example

Calculate the length of each word in the tuple:

```
Def myFunc(s):
      return(len(s))
X = map(myFunc,('Apple','Banana','kiwi'))
Print(list(X))

X = list(map(lambda s: len(s) ,('Apple','Banana','kiwi')))
print(X)

X=[len(s)  for s in ('Apple','Banana','kiwi')]
print(X)
```

filter() Function

filter(func, seq)

- It offers an elegant way to filter out all the elements of a sequence "seq", for which the function func returns True. i.e. an item will be produced by the iterator result of filter(func, seq) if item is included in the sequence "seq" and if func(item) returns True.
- In other words: The function filter(f,l) needs a function f as its first argument. f has to return a Boolean value, i.e. either True or False. This function will be applied to every element of the list l. Only if f returns True will the element be produced by the iterator, which is the return value of filter(function, sequence).

filter() Function

In the following example, we filter out first the odd and then the even elements of the sequence of the first 11 Fibonacci numbers:

```
fibonacci = [0,1,1,2,3,5,8,13,21,34,55]
odd_num = list(filter(lambda x: x%2, fibonacci ))
Print(odd_num)

even_num = list(filter(lambda x: x%2==0, fibonacci ))
Print(even_num)

odd_num = [x for x in fibonacci if x%2]
even_num = [x for x in fibonacci if x%2==0]
```


reduce() Function

The **reduce(fun,seq)** function is used to **apply a particular function passed in its argument to all of the list elements** mentioned in the sequence passed along. This function is defined in "**functools**" module.

It performs a rolling-computation as specified by the passed function to the neighboring elements, by taking a *function* and an *iterable* as arguments, and returns the *final* computed value.

Working :

1. At first step, first two elements of sequence are picked and the result is obtained.
2. Next step is to apply the same function to the previously attained result and the number just succeeding the second element and the result is again stored.
3. This process continues till no more elements are left in the container.
4. The final returned result is returned and printed on console.

reduce() Function

Example:

```
from functools import reduce
# Returns the sum of two elements
def sumTwo(a,b):
   return a+b

result = reduce(sumTwo, [1, 2, 3, 4])
print(result)

from functools import reduce
# Returns the sum of all the elements using `reduce`
result = reduce((lambda a, b: a + b), [1, 2, 3, 4])
print(result)
```


Comparison of Lambda and List Comprehension

- List Comprehension is used to create lists, Lambdas are functions that can process like other functions and thus return values or list.
- Lambda function process is the same as other functions and returns the value of the list. The **Lambda** function itself cannot be used to iterate through a list. It return a list with the help of **map()** and **list()** functions.

  list(map(lambda argument: manipulate(argument), iterable))
- List comprehension performance is better than lambda because filter() in lambda is slower than list comprehension.

NumPy Library

- NumPy, which stands for Numerical Python, is a library consisting of multidimensional array objects and a collection of routines for processing those arrays.
- NumPy is a general-purpose array-processing package. It provides a high-performance multidimensional array object, and tools for working with these arrays : the **n-dimensional array**. This is simple yet powerful data structure.
- In Python we have lists that serve the purpose of arrays, but they are slow to process. NumPy aims to provide an array object that is up to 50x faster than traditional Python lists.
- The array object in NumPy is called ***ndarray*** it provides a lot of supporting functions that make working with ***ndarray*** very easy.
- Arrays are very frequently used in data science, where speed and resources are very important.

NumPy Library continued ….

- It is the fundamental package for scientific computing with Python. It contains various features including these important ones:
- A powerful N-dimensional array object
- Sophisticated (broadcasting) functions
- Tools for integrating C/C++ and Fortran code
- Useful linear algebra, Fourier transform, and random number capabilities

Besides its obvious scientific uses, NumPy can also be used as an efficient multi-dimensional container of generic data.

Arbitrary data-types can be defined using Numpy which allows NumPy to seamlessly and speedily integrate with a wide variety of databases.

Why is NumPy Faster Than Lists?

- NumPy arrays are stored at one continuous place in memory unlike lists, so processes can access and manipulate them very efficiently.
- This behavior is called locality of reference in computer science.
- This is the main reason why NumPy is faster than lists. Also it is optimized to work with latest CPU architectures.

NumPy is a Python library and is written partially in Python, but most of the parts that require fast computation are written in C or C++.

Arrays in NumPy

NumPy's main object is the homogeneous multidimensional array.

- It is a table of elements (usually numbers), all of the same type, indexed by a tuple of positive integers.
- In NumPy dimensions are called *axes*. The number of axes is *rank*.
- NumPy's array class is called **ndarray**. It is also known by the alias **array**.
- Items in the collection can be accessed using a zero-based index.
- Every item in an ndarray takes the same size of block in the memory.

Example:

```
import numpy
arr = numpy.array([1, 2, 3, 4, 5])
print(arr)
```

Array creation:

There are various ways to create arrays in NumPy.

- we can pass a list, tuple or any array-like object into the array() method, and it will be converted into an ndarray.

General Syntax :

numpy.array(object, dtype = None, copy = True, order = None, subok = False, ndmin = 0)

1	object	It represents the collection object. It can be a list, tuple, dictionary, set, etc.
2	dtype	We can change the data type of the array elements by changing this option to the specified type. The default is none.
3	copy	It is optional. By default, it is true which means the object is copied.
4	order	There can be 3 possible values assigned to this option. It can be C (column order), R (row order), or A (any)
5	subok	The returned array will be base class array by default. We can change this to make the subclasses passes through by setting this option to true.
6	ndmin	It represents the minimum dimensions of the resultant array.

Array creation:

Example:

```
import numpy as np

arr = np.array( [[ 1, 2, 3],
                 [ 4, 2, 5]] )

print("Array is of type: ", type(arr))

print("No. of dimensions: ", arr.ndim)

print("Shape of array: ", arr.shape)

print("Size of array: ", arr.size)

print("Array stores elements of type: ", arr.dtype)
```


Data Types & Description

Sr.No.	Data Types & Description
1.	bool_ Boolean (True or False) stored as a byte
2.	int_ Default integer type (same as C long; normally either int64 or int32)
3.	Intc Identical to C int (normally int32 or int64)
4.	int8 Byte (-128 to 127), int16 Integer (-32768 to 32767), int32 Integer (-2147483648 to 2147483647), int64 Integer (-9223372036854775808 to 9223372036854775807)
5.	uint8 Unsigned integer (0 to 255), uint16 Unsigned integer (0 to 65535), uint32 Unsigned integer (0 to 4294967295), uint64 Unsigned integer (0 to 18446744073709551615)
6.	float_ Shorthand for float64, float16 Half precision float: sign bit, 5 bits exponent, 10 bits mantissa. float32 Single precision float: sign bit, 8 bits exponent, 23 bits mantissa, float64 Double precision float: sign bit, 11 bits exponent, 52 bits mantissa
7.	complex_ Shorthand for complex128, complex64 Complex number, represented by two 32-bit floats (real and imaginary components), complex128 Complex number, represented by two 64-bit floats (real and imaginary components)

Data Types & Description

- Each built-in data type has a character code that uniquely identifies it.
- **'b'** – boolean
- **'i'** – (signed) integer
- **'u'** – unsigned integer
- **'f'** – floating-point
- **'c'** – complex-floating point
- **'m'** – timedelta
- **'M'** – datetime
- **'O'** – (Python) objects
- **'S', 'a'** – (byte-)string
- **'U'** – Unicode
- **'V'** – raw data (void)

Try This :

```
import numpy as np
student = np.dtype([('name','S20'), ('age', 'i1'), ('marks', 'f4')])
a = np.array([('abc', 21, 50),('xyz', 18, 75)], dtype = student)
print a
```


Dimensions in Arrays

0-D Arrays

- 0-D arrays, or Scalars, are the elements in an array. Each value in an array is a 0-D array.

```
import numpy as np
arr = np.array(42)
print(arr)
```

1-D Arrays

- An array that has 0-D arrays as its elements is called uni-dimensional or 1-D array.
- These are the most common and basic arrays.

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
print(arr)
```


Dimensions in Arrays

2-D Arrays

- An array that has 1-D arrays as its elements is called a 2-D array.
- These are often used to represent matrix or 2nd order tensors.

```
import numpy as np
arr = np.array([[1, 2, 3], [4, 5, 6]])
print(arr)
```

3-D arrays

- An array that has 2-D arrays (matrices) as its elements is called 3-D array.
- These are often used to represent a 3rd order tensor.

```
import numpy as np
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])
print(arr)
```

Array Creation Contitues …

- Often, the elements of an array are originally unknown, but its size is known. Hence, NumPy offers several functions to create arrays with **initial placeholder content**. These minimize the necessity of growing arrays, an expensive operation.
  For example: np.zeros, np.ones, np.full, np.empty, etc.

```
c = np.zeros((3, 4))
print ("\nAn array initialized with all zeros:\n", c)

d = np.full((3, 3), 6, dtype = 'complex')
print ("\nAn array initialized with all 6s Array type is complex:\n", d)
```

- To create sequences of numbers, NumPy provides a function analogous to range that returns arrays instead of lists.
  - **arange:** returns evenly spaced values within a given interval. **step** size is specified.
  - **linspace:** returns evenly spaced values within a given interval. **num** no. of elements are returned.

```
f = np.arange(0, 30, 5)
g = np.linspace(0, 5, 10)
```


Array Creation Contitues …

- **Reshaping array:** We can use **reshape** method to reshape an array. Consider an array with shape (a1, a2, a3, …, aN). We can reshape and convert it into another array with shape (b1, b2, b3, …, bM). The only required condition is:
  a1 x a2 x a3 … x aN = b1 x b2 x b3 … x bM . (i.e original size of array remains unchanged.)

```
arr = np.array([[1, 2, 3, 4],
                [5, 2, 4, 2],
                [1, 2, 0, 1]])

newarr = arr.reshape(2, 2, 3)

print ("\nOriginal array:\n", arr)
print ("Reshaped array:\n", newarr)
```


Array Creation Contitues …

- **Flatten array:** We can use **flatten** method to get a copy of array collapsed into **one dimension**. It accepts *order* argument. Default value is 'C' (for row-major order). Use 'F' for column major order.

```
arr = np.array([[1, 2, 3], [4, 5, 6]])
flarr = arr.flatten()

print ("\nOriginal array:\n", arr)
print ("Fattened array:\n", flarr)
```

numpy.empty

It creates an uninitialized array of specified shape and dtype. It uses the following constructor −

numpy.empty(shape, dtype = float, order = 'C')

The constructor takes the following parameters.

Sr.No.	Parameter & Description
1	**Shape** Shape of an empty array in int or tuple of int
2	**Dtype** Desired output data type. Optional
3	**Order** 'C' for C-style row-major array, 'F' for FORTRAN style column-major array

```
import numpy as np
x = np.empty([3,2], dtype = int)
print x
```


NumPy - Indexing & Slicing

- Contents of ndarray object can be accessed and modified by indexing or slicing, just like Python's in-built container objects.
- Items in ndarray object follows zero-based index. Three types of indexing methods are available − **field access, basic slicing** and **advanced indexing**.
- Basic slicing is an extension of Python's basic concept of slicing to n dimensions. A Python slice object is constructed by giving **start, stop**, and **step** parameters to the built-in **slice** function. This slice object is passed to the array to extract a part of array.

```
import numpy as np
a = np.arange(10)
s = slice(2,7,2)
print(a)
print a[s]
```


NumPy - Indexing & Slicing

The same result can also be obtained by giving the slicing parameters separated by a colon : (start:stop:step) directly to the **ndarray** object.

```
import numpy as np
a = np.arange(10)
b = a[2:7:2]
print b
```

- If only one parameter is put, a single item corresponding to the index will be returned.
- If a : is inserted in front of it, all items from that index onwards will be extracted.
- If two parameters (with : between them) is used, items between the two indexes (not including the stop index) with default step one are sliced.

Try This: a[5:8]=12

```
print (a)
```

NumPy - Indexing & Slicing

With higher dimensional arrays, you have many more options. In a two-dimensional array, the elements at each index are no longer scalars but rather one-dimensional arrays:

```
arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(arr2d[2])
```

We can pass a comma-separated list of indices to select individual elements.

```
print(arr2d[0][2])
Or
print(arr2d[0,2])
```


NumPy - Indexing & Slicing

In multidimensional arrays, if you omit later indices, the returned object will be a lower-dimensional ndarray consisting of all the data along the higher dimensions. So in the 2X2X3 array:

```
arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
print(arr3d)
print(arr3d[0])
print(arr3d[1,0])
```

- Arr3d[0] is a 2X3 array
- Similarly, arr3d[1,0] gives you all of the values whose indices start with (1,0) forming a 1-dimensional array

Array Operations

Scalar Addition

Scalars can be added and subtracted from arrays and arrays can be added and subtracted from each other:

```
import numpy as np

a = np.array([1, 2, 3])
b = a + 2
print(b)
```

Array Operations

Scalar Multiplication

NumPy arrays can be multiplied and divided by scalar integers and floats:

```
a = np.array([1,2,3])
b = 3*a
print(b)

a = np.array([10,20,30])
b = a/2
print(b)
```


Element-wise Operations

Arrays enable you to perform mathematical operations on whole blocks of data using similar syntax to the equivalent operations between scalar elements.

Operations between differently sized arrays is called ***broadcasting***. Input arrays for performing arithmetic operations such as add(), subtract(), multiply(), and divide() must be either of the same shape or should conform to array broadcasting rules.

Example : Array-Operations.py

Element-wise Operations
(*Unary functions*)

Function	Description
abs, fabs	Compute the absolute value element-wise for integer, floating point, or complex values. Use fabs as a faster alternative for non-complex-valued data
sqrt	Compute the square root of each element. Equivalent to arr ** 0.5
square	Compute the square of each element. Equivalent to arr ** 2
exp	Compute the exponent e^x of each element
log, log10, log2, log1p	Natural logarithm (base e), log base 10, log base 2, and log(1 + x), respectively
sign	Compute the sign of each element: 1 (positive), 0 (zero), or -1 (negative)
ceil	Compute the ceiling of each element, i.e. the smallest integer greater than or equal to each element
floor	Compute the floor of each element, i.e. the largest integer less than or equal to each element
rint	Round elements to the nearest integer, preserving the dtype
modf	Return fractional and integral parts of array as separate array

Element-wise Operations
(Unary functions)

Example:

```
import numpy as np

arr = np.arange(10)

print(arr)
print(np.sqrt(arr))
print(np.exp(arr))
```


Element-wise Operations
(Binary universal functions)

Function	Description
add	Add corresponding elements in arrays
subtract	Subtract elements in second array from first array
multiply	Multiply array elements
divide, floor_divide	Divide or floor divide (truncating the remainder)
power	Raise elements in first array to powers indicated in second array
maximum, fmax	Element-wise maximum. fmax ignores NaN
minimum, fmin	Element-wise minimum. fmin ignores NaN
mod	Element-wise modulus (remainder of division)
greater, greater_equal, less, less_equal, equal, not_equal	Perform element-wise comparison, yielding boolean array. Equivalent to infix operators >, >=, <, <=, ==, !=

Array Operations

numpy.reciprocal()

This function returns the reciprocal of argument, element-wise. For elements with absolute values larger than 1, the result is always 0 because of the way in which Python handles integer division. For integer 0, an overflow warning is issued.

```
import numpy as np
a = np.array([0.25, 1.33, 1, 0, 100])
print(a )
print ('\n' )
print (np.reciprocal(a) )

b = np.array([100], dtype = int)
print( b )
print( np.reciprocal(b) )
```

Array Operations

numpy.power()

This function treats elements in the first input array as base and returns it raised to the power of the corresponding element in the second input array.

```
import numpy as np
a = np.array([10,100,1000])
print( a)
print (np.power(a,2) )
print ('\n'  )

b = np.array([1,2,3])
print( b )
print ('\n')
print (np.power(a,b))
```


Array Operations

numpy.mod()

- This function returns the remainder of division of the corresponding elements in the input array. The function **numpy.remainder()** also produces the same result.

```
import numpy as np
a = np.array([10,20,30])
b = np.array([3,5,7])

print ('First array:',a,'\n' )

print ('Second array:',b,'\n' )

print ('Applying mod() function:',np.mod(a,b),'\n')

print ('Applying remainder() function:',np.remainder(a,b),'\n' )
```


Aggregate and Statistical Functions in Numpy

In the Python numpy module, we have many aggregate functions or statistical functions to work with a single-dimensional or multi-dimensional array.

The Python numpy aggregate functions are sum, min, max, mean, average, product, median, standard deviation, variance, argmin, argmax, percentile etc.

Example: Array-Aggregate-Functions.py

Aggregate and Statistical Functions in Numpy

Some of the aggregate and statistical functions are given below:

- **np.sum(m)**: Used to find out the **sum** of the given array.
- **np.prod(m)**: Used to find out the **product(multiplication)** of the values of m.
- **np.mean(m)**: It returns the **mean** of the input array m.
- **np.std(m)**: It returns the **standard deviation** of the given input array m.
- **np.var(m)**: Used to find out the **variance** of the data given in the form of array m.
- **np.min(m)**: It returns the **minimum value** among the elements of the given array m.
- **np.max(m)**: It returns the **maximum value** among the elements of the given array m.
- **np.argmin(m)**: It returns the **index of the minimum value** among the elements of the array m.
- **np.argmax(m)**: It returns the **index of the maximum value** among the elements of the array m.
- **np.median(m)**: It returns the **median** of the elements of the array m.

Insert Row / Column in Array

Numpy.hstack is a function in Python that is used to horizontally stack sequences (column-wise) of input arrays in order to make a single array. With hstack() function, you can append data horizontally. It is a very convenient function in NumPy.

Example:

```
import numpy as np
x = np.array((3,5,7))
y = np.array((5,7,9))
print(np.hstack((x,y)))

x = np.array([[3], [5], [7]])
y = np.array([[5], [7], [9]])
print(np.hstack((x,y)))
```


Insert Row / Column in Array

Try for following outputs:

```
[[3 1 5 2]
 [5 3 7 4]
 [7 5 9 6]]


[[3 1 5]
 [5 2 7]
 [7 3 9]]
```

Insert Row / Column in Array

The vstack() function is used to stack arrays in sequence vertically (row wise).

Example :

```
x = np.array([3, 5, 7])
y = np.array([5, 7, 9])
print(np.vstack((x,y)))

x = np.array([[3], [5], [7]])
y = np.array([[5], [7], [9]])
print(np.vstack((x,y)))
```


Append Row / Column in Array

You can add a NumPy array element by using the append() method of the NumPy module.

The syntax of append is as follows:

```
numpy.append(array, value, axis)
```

- The values will be appended at the end of the array and a new ndarray will be returned with new and old values.
- The axis is an optional integer along which define how the array is going to be displayed. If the axis is not specified, the array structure will be flattened as you will see later.

```
a = np.array([1, 2, 3])
newArray = np.append (a, [10, 11, 12])
print(newArray)
```


Append Row / Column in Array

Add a column

- We can use the append() method of NumPy to insert a column.
- Consider the example below where we created a 2-dimensional array and inserted two columns:

```
a = np.array([[1, 2, 3], [4, 5, 6]])
b = np.array([[400], [800]])
newArray = np.append(a, b, axis=1)
print(newArray)
```

Append Row / Column in Array

Append a row

- In this section, we will be using the append() method to add a row to the array. It's as simple as appending an element to the array. Consider the following example:

```
a = np.array([[1, 2, 3], [4, 5, 6]])
newArray = np.append(a, [[50, 60, 70]], axis = 0)
print(newArray)
```


Append Row / Column in Array

In NumPy, we can also use the insert() method to insert an element or column. The difference between the insert() and the append() method is that we can specify at which index we want to add an element when using the insert() method but the append() method adds a value to the end of the array.

Consider the example below:

```
a = np.array([1, 2, 3])
newArray = np.insert(a, 1, 90)
print(newArray)
```

Here the insert() method adds the element at index 1. Remember the array index starts from 0.

Few Useful Methods

Check if NumPy array is empty

We can use the size method which returns the total number of elements in the array.

```
a = np.array([1, 2, 3])
if(a.size == 0):
   print("The given Array is empty")
else:
   print("The array = ", a)
```

Check with

```
a = np.array([])
```

Few Useful Methods

Find the index of a value

To find the index of value, we can use the where() method of the NumPy module

```
a = np.array([1, 2, 3, 4, 5])
print("5 is found at index: ", np.where(a == 5))
```

The where() method will also return the datatype. If you want to just get the index, use the following code:

```
a = np.array([1, 2, 3, 4, 5])
index = np.where(a == 5)
print("5 is found at index: ", index[0])
```


Few Useful Methods

NumPy array to CSV

To export the array to a CSV file, we can use the savetxt() method of the NumPy module as illustrated in the example below:

```
a = np.array([1, 2, 3, 4, 5])
np.savetxt("D:/Python Programming/Scripts/myArray.csv", a)
```

This code will generate a CSV file in the location where our Python code file is stored. You can also specify the path.

Sort NumPy array

- You can sort NumPy array using the sort() method of the NumPy module:
- The sort() function takes an optional axis (an integer) which is -1 by default. The axis specifies which axis we want to sort the array. -1 means the array will be sorted according to the last axis.

```
print("Sorted array = ", np.sort(a))
```


Pandas Library

pandas is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language.

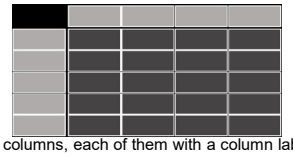
- Pandas is a Python library used for working with data sets.
- It has functions for analyzing, cleaning, exploring, and manipulating data.
- The name "Pandas" has a reference to both "Panel Data", and "Python Data Analysis" and was created by Wes McKinney in 2008.

```
import pandas as pd
df = pd.read_csv('D:/Python Programming/Scripts/myArray.csv')
print(df.to_string())
```

Data -Frames

A **DataFrame** is a 2-dimensional data structure that can store data of different types (including characters, integers, floating point values, categorical data and more) in columns. It is similar to a spreadsheet, a SQL table

Column



←Row

The table has 3 columns, each of them with a column label. The column labels are respectively Name, Age and Sex.

The column Name consists of textual data with each value a string, the column Age are numbers and the column Sex is textual data.

Data -Frames

```
df = pd.DataFrame(
{
  "Name": [
  "Braund, Mr. Owen Harris",
  "Allen, Mr. William Henry",
  "Bonnell, Miss. Elizabeth",
  ],
  "Age": [22, 35, 58],
  "Sex": ["male", "male", "female"],
}
)
print(df)
```


Data -Frames

Locate Row

Pandas use the **loc** attribute to return one or more specified row(s)

```
print(df.loc[0])
```

This returns a Pandas **Series**.

```
print(df.loc[[0, 1]])
```

When using [ ], the result is a Pandas **DataFrame**.

Data -Frames

Named Indexes

With the **index** argument, you can name your own indexes.

```
data = {
  "calories": [420, 380, 390],
  "duration": [50, 40, 45]
}
df = pd.DataFrame(data, index = ["day1", "day2", "day3"])
print(df)
```

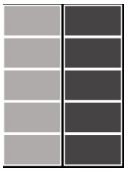
Locate Named Indexes

Use the named index in the **loc** attribute to return the specified row(s).

```
print(df.loc["day2"])
```


Series

- A Pandas Series is like a column in a table.
- It is a one-dimensional array holding data of any type.
- A pandas Series has no column labels, as it is just a single column of a DataFrame. A Series does have row labels.
- If nothing else is specified, the values are labeled with their index number. First value has index 0, second value has index 1 etc.
- This label can be used to access a specified value.
- With the **index** argument, we can give labels.



```
a = [1, 7, 2]
myvar = pd.Series(a, index = ["x", "y", "z"])
print(myvar)
```


Series

When selecting a single column of a pandas **DataFrame**, the result is a pandas **Series**. To select the column, use the column label in between square brackets [].

Example :

```
df["Age"]
```

Or

```
ages = pd.Series([22, 35, 58], name="Age")
print(ages)
```

Series

Key/Value Objects as Series

You can also use a key/value object, like a dictionary, when creating a Series.

```
import pandas as pd
calories = {"day1": 420, "day2": 380, "day3": 390}
myvar = pd.Series(calories)
print(myvar)
```

The keys of the dictionary become the labels.

To select only some of the items in the dictionary, use the **index** argument and specify only the items you want to include in the Series.

```
import pandas as pd
calories = {"day1": 420, "day2": 380, "day3": 390}
myvar = pd.Series(calories, index = ["day1", "day2"])
print(myvar)
```


Data Preparation and Pre-Processing

We can equate data preparation with the framework of the KDD Process -- specifically the first 3 major steps -- which re **selection**, **preprocessing**, and **transformation**.

1. **Loading data**

The first step for data preparation is to get some data. If you have a .csv file, you can easily load it up in your system using the **read_csv()** function in pandas. We can work with Data-frames and Series as well.

```
df = pd.read_csv('D:/Python Programming/Scripts/myArray.csv')
```


Data Preparation and Pre-Processing

2. **Missing Data**

– **Handling Missing Data**

Missing data can arise in the dataset due to multiple reasons: the data for the specific field was not added by the user/data collection application, data was lost while transferring manually, a programming error, etc.

For numerical data, pandas uses a floating point value **NaN** (Not a Number) to represent missing data. It is a unique value defined under the library **Numpy** so we will need to import it as well. NaN is the default missing value marker for reasons of computational speed and convenience. This is a sentinel value, in the sense that it is a dummy data or flag value that can be easily detected and worked with using functions in pandas.

```
data = pd.Series([0, 1, 2, 3, 4, 5, np.nan, 6, 7, 8])
data.isnull()
```

We used the function **isnull()** which returns a boolean true or false value. True, when the data at that particular index is actually missing or NaN. The opposite of this is the **notnull()** function.

Data Preparation and Pre-Processing

Furthermore, we can use the **dropna()** function to filter out missing data and to remove the null (missing) value and see only the non-null values. However, the NaN value is not really deleted and can still be found in the original dataset.

What you can do to really "drop" or delete the NaN value is either store the new dataset (without NaN) so that the original data Series is not tampered or apply a drop **inplace**. The **inplace** argument has a default value of false.

```
not_null_data = data.dropna()
print(not_null_data)

data.dropna(inplace = True)
print(data)
```


Data Preparation and Pre-Processing

Now we try it on Data-frames:

```
data_dim =
pd.DataFrame([[1,2,3,np.nan],[4,5,np.nan,np.nan],[7,np.nan,np.nan,np.nan],[np.nan
,np.nan,np.nan,np.nan]])
print(data_dim)
```

Now let's say we only want to drop rows or columns that are all null or only those that contain a certain amount of null values.

Try **data_dim.dropna() :** It will not work and the real dataset is not tampered.

Now, try **data_dim.dropna(how = 'all')**

Also try **data_dim.dropna(axis = 1, thresh = 2)**

Data Preparation and Pre-Processing

– **Filling in Missing Data**

To replace or rather "fill in" the null data, you can use the **fillna()** function. For example, let's try to use the same dataset as above and try to fill in the NaN values with 0.

```
data_dim_fill = data_dim.fillna(0)
print(data_dim_fill)
```

And like with **dropna()** you can also do many other things depending on the kind of argument you pass. Also a reminder that passing the **inplace=true** argument will make the change to the original dataset.

We can pass a dictionary to use different values for each column:

```
data_dim_fill = data_dim.fillna({0: 0, 1: 8, 2: 9, 3: 10})
print(data_dim_fill)
```

DataFrame Indexing

- Indexing in pandas means simply selecting particular rows and columns of data from a DataFrame.
- Indexing could mean selecting all the rows and some of the columns, some of the rows and all of the columns, or some of each of the rows and columns.
- Indexing can also be known as **Subset Selection**.
- The Python and NumPy indexing operators "[ ]" and attribute operator "." provide quick and easy access to Pandas data structures across a wide range of use cases.
- But, since the type of the data to be accessed isn't known in advance, directly using standard operators has some optimization limits.
- We take advantage of some optimized pandas data access methods like **.loc(), .iloc(), .ix().**

DataFrame Indexing

loc() Method: (Label Based)

- Pandas provide various methods to have purely **label based indexing**. When slicing, the start bound is also included. Integers are valid labels, but they refer to the label and not the position.

.loc() has multiple access methods like −

- A single scalar label
- A list of labels
- A slice object
- A Boolean array

loc takes two single/list/range operator separated by ','. The first one indicates the row and the second one indicates columns.

DataFrame Indexing

Example:

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(6, 4),
index = ['a','b','c','d','e','f'], columns = ['A', 'B', 'C','D'])
print (df.loc['a':'d'])
print (df.loc['a':'c','D'])
print (df.loc['a':'f','A':'B'])
```

DataFrame Indexing

.iloc() method:

- Pandas provide various methods in order to get purely integer based indexing. Like python and numpy, these are **0-based** indexing.
- It is primarily integer position based from 0 to length – 1 of the axis

The various access methods are as follows −

- An Integer
- A list of integers
- A range of values

DataFrame Indexing

Example:

```
import pandas as pd
import numpy as np

df1 = pd.DataFrame(np.random.randn(8, 3),columns = ['A', 'B', 'C'])
print (df1.iloc[:8])
print (df1.iloc[:4])
print (df1.iloc[2:4, 1:3])
```


DataFrame Indexing

.ix() method:

- Besides pure label based and integer based, Pandas provides a hybrid method for selections and subsetting the object using the .ix() operator.
- The .ix indexer is deprecated in all the version after 0.20.0, in favor of the more strict .iloc and .loc indexers.

```
df = pd.DataFrame(np.random.randn(8, 4), columns = ['A', 'B', 'C', 'D'])
print (df.ix[:,'A'])
```

DataFrame Indexing

Select Data Using Columns

In addition to location-based and label-based indexing, you can also select data from **pandas** dataframes by selecting entire columns using the column names.

dataframe[“column”]

Above command provides the data from the column as a **pandas** series, which is a one-dimensional array. A **pandas** series is useful for selecting columns for plotting using **matplotlib**.

You can also specify that you want an output that is also a **pandas** dataframe.

dataframe[[“column”]]

which includes a second set of brackets [ ], to indicate that the output should be a **pandas** dataframe.

DataFrame Indexing

You can also select all data from multiple columns in a **pandas** dataframe using:

dataframe[[“column1”, “column2”]]

Since the results of your selection are also a **pandas** dataframe, you can assign the results to a new **pandas** dataframe.

Try This:

Use avg-precip-months.csv

create a new **pandas** dataframe that only contains the ***months*** and ***seasons*** column effectively dropping the ***precip*** values.

DataFrame Indexing

Filter Data Using Specific Values

In addition to location-based and label-based indexing, you can select or filter data based on specific values within a column using:

dataframe[dataframe[“column1”]==value]

This will return all rows containing that value within the specified column.

Again, you can also save the output to a new dataframe by setting it equal to the output of the filter.

You can also filter using a **comparison operator** on numeric values.

Try This :

Select all rows that have a season value of summer.

The query() Method

The query() Method

- Python is a great language for doing data analysis, primarily because of the fantastic ecosystem of data-centric Python packages. ***Pandas*** is one of those packages that makes importing and analyzing data much easier.
- Analyzing data requires a lot of filtering operations. Pandas provide many methods to filter a Data frame and dataframe.query() is one of them.
- DataFrame objects have a query() method that allows selection using an expression. You can get the value of the frame where column b has values between the values of columns a and c.

The query() Method

- ***Syntax:*** *DataFrame.query(expr, inplace=False, **kwargs)*
- ***Parameters:***
  expr: *Expression in string form to filter data.*
  inplace: *Make changes in the original data frame if True*
  kwargs: *Other keyword arguments.*
- ***Return type:*** *Filtered Data frame*

Note: dataframe.query() method only works if the column name doesn't have any empty spaces. So before applying the method, spaces in column names are replaced with '_'

The query() Method

Example #1: Single condition filtering (Employee.csv)

In this example, the data is filtered on the basis of single condition. Before applying the query() method, the spaces in column names have been replaced with '_'.

```
import pandas as pd
import numpy as np
# making data frame from csv file
data1 = pd.read_csv('D:/Python Programming/Scripts/employees.csv')
 # replacing blank spaces with '_'
data1.columns =[column.replace(" ", "_") for column in data1.columns]
 # filtering with query method
data1.query('Senior_Management == True', inplace = True)

# display
print(data1)
```

The query() Method

Example #2: Multiple condition filtering

```
# making data frame from csv file
data = pd.read_csv('D:/Python Programming/Scripts/employees.csv')

# replacing blank spaces with '_'
data.columns =[column.replace(" ", "_") for column in data.columns]

# filtering with query method
data.query('Senior_Management == True and Gender =="Male" and
Team =="Marketing" and First_Name =="Johnny"', inplace = True)

# display
print(data)
```


Data Visualization

Data visualization is the discipline of trying to understand data by placing it in a visual context so that patterns, trends and correlations that might not otherwise be detected can be exposed.

Data Visualization is the presentation of data in graphical format. It helps people understand the significance of data by summarizing and presenting huge amount of data in a simple and easy-to-understand format and helps communicate information clearly and effectively.

Data visualization in python is perhaps one of the most utilized features for data science with python in today's day and age. The libraries in python come with lots of different features that enable users to make highly customized, elegant, and interactive plots.

Data Visualization

Useful packages for visualizations in python

- **Matplotlib**
- Matplotlib is a visualization library in Python for 2D plots of arrays. Matplotlib is written in Python and makes use of the NumPy library. Matplotlib comes with a wide variety of plots like line, bar, scatter, histogram, etc. which can help us, deep-dive, into understanding trends, patterns, correlations. It was introduced by John Hunter in 2002.
- **Seaborn**
- Seaborn is a dataset-oriented library for making statistical representations in Python. It is developed atop matplotlib and to create different visualizations. It is integrated with pandas data structures. The library internally performs the required mapping and aggregation to create informative visuals It is recommended to use a Jupyter/IPython interface in matplotlib mode.

Data Visualization

- **Bokeh**
- Bokeh is an interactive visualization library for modern web browsers. It is suitable for large or streaming data assets and can be used to develop interactive plots and dashboards. There is a wide array of intuitive graphs in the library which can be leveraged to develop solutions. It works closely with PyData tools. The library is well-suited for creating customized visuals according to required use-cases.
- **Altair**
- Altair is a declarative statistical visualization library for Python. Altair's API is user-friendly and consistent and built atop Vega-Lite JSON specification. Declarative library indicates that while creating any visuals, we need to define the links between the data columns to the channels (x-axis, y-axis, size, color). With the help of Altair, it is possible to create informative visuals with minimal code.

Data Visualization

- **plotly**
- plotly.py is an interactive, open-source, high-level, declarative, and browser-based visualization library for Python. It holds an array of useful visualization which includes scientific charts, 3D graphs, statistical charts, financial charts among others. Plotly graphs can be viewed in Jupyter notebooks, standalone HTML files, or hosted online.
- **ggplot**
- ggplot is a Python implementation of the grammar of graphics. The Grammar of Graphics refers to the mapping of data to aesthetic attributes (colour, shape, size) and geometric objects (points, lines, bars). The basic building blocks according to the grammar of graphics are data, geom (geometric objects), stats (statistical transformations), scale, coordinate system, and facet.

Types of Charts for Analyzing & Presenting Data

- **Histogram**
  The histogram represents the frequency of occurrence of specific phenomena which lie within a specific range of values and arranged in consecutive and fixed intervals.

- **Bar Charts**
  A bar chart can be created using the bar method. The bar-chart isn't automatically calculating the frequency of a category so we are going to use pandas value_counts function to do this. The bar-chart is useful for categorical data that doesn't have a lot of different categories (less than 30) because else it can get quite messy.

Example: Test-plot.py

Types of Charts for Analyzing & Presenting Data

- **Pie Chart :**
  A pie chart shows a static number and how categories represent part of a whole the composition of something. A pie chart represents numbers in percentages, and the total sum of all segments needs to equal 100%.

- **Scatter plot :**
- A scatter chart shows the relationship between two different variables and it can reveal the distribution trends. It should be used when there are many different data points, and you want to highlight similarities in the data set. This is useful when looking for outliers and for understanding the distribution of your data.

Example: Piechart.py

Creating Pie Charts

```
import matplotlib.pyplot as plt
import numpy as np

y = np.array([35, 25, 25, 15])

plt.pie(y)
plt.show()
```

- As you can see the pie chart draws one piece (called a wedge) for each value in the array (in this case [35, 25, 25, 15]).
- By default the plotting of the first wedge starts from the x-axis and move *counterclockwise*:

Creating Pie Charts

Labels

```
mylabels = ["Apples", "Bananas", "Cherries", "Dates"]
plt.pie(y, labels = mylabels)
plt.show()
```

Explode

- Maybe you want one of the wedges to stand out? The explode() parameter allows you to do that.
- The explode() parameter, if specified, and not None, must be an array with one value for each wedge.
- Each value represents how far from the center each wedge is displayed:

```
myexplode = [0.2, 0, 0, 0]
plt.pie(y, labels = mylabels, explode = myexplode, shadow=true)
plt.show()
```

Colors and legends

Color

You can set the color of each wedge with the colors parameter.

The colors parameter, if specified, must be an array with one value for each wedge:

```
mycolors = ["black", "hotpink", "b", "#4CAF50"]
plt.pie(y, labels = mylabels, colors = mycolors)
plt.show()
```

Legend

To add a list of explanation for each wedge, use the legend() function:

```
y = np.array([35, 25, 25, 15])
mylabels = ["Apples", "Bananas", "Cherries", "Dates"]

plt.pie(y, labels = mylabels)
plt.legend()
plt.show()
```


Legend With Header

Legend With Header

To add a header to the legend, add the title parameter to the legend():

```
y = np.array([35, 25, 25, 15])
mylabels = ["Apples", "Bananas", "Cherries", "Dates"]

plt.pie(y, labels = mylabels)
plt.legend(title = "Four Fruits:")
plt.show()
```

Change Figure Size in Matplotlib

```
plt.figure(figsize=(width, height))

plt.figure(figsize=(3, 3))
```


Plot() Method

```
import matplotlib.pyplot as plt
import numpy as np

xpoints = np.array([0, 6])
ypoints = np.array([0, 250])

plt.plot(xpoints, ypoints)
plt.plot(xpoints, ypoints, marker= 'o')
plt.show()
```

Other possible markers are - *, . , x, X, +, P, s (Square), D (Diamond), v (Triangle Down), ^ (Triangle up), < (Triangle left), > (Triangle right), _ (Hline), | (Vline)

Plot() Method

Line Reference

Line Syntax	Description
'-'	Solid line
':'	Dotted line
'--'	Dashed line
'-.'	Dashed/dotted line

```
ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, 'o:r')
plt.show()
```


Plot() Method

Color Reference

Color Syntax	Description
'r'	Red
'g'	Green
'b'	Blue
'c'	Cyan
'm'	Magenta
'y'	Yellow
'k'	Black
'w'	White

Plot() Method

```
import matplotlib.pyplot as plt
import numpy as np

xpoints = np.array([1, 2, 6, 8])
ypoints = np.array([3, 8, 1, 10])

plt.plot(xpoints, ypoints)
plt.show()
```

If we do not specify the points in the x-axis, they will get the default values 0, 1, 2, 3, (etc. depending on the length of the y-points.

Marker Size - plt.plot(ypoints, marker = 'o', ms = 20)

Multiple Lines in one figure

Line Width - plt.plot(ypoints, linewidth = '20.5')

Multiple Lines

```
import matplotlib.pyplot as plt
import numpy as np

y1 = np.array([3, 8, 1, 10])
y2 = np.array([6, 2, 7, 11])

plt.plot(y1)
plt.plot(y2)

plt.show()
```


Create Labels for a Plot

Create Labels for a Plot

```
import numpy as np
import matplotlib.pyplot as plt

x = np.array([80, 85, 90, 95, 100, 105, 110, 115, 120, 125])
y = np.array([240, 250, 260, 270, 280, 290, 300, 310, 320, 330])

plt.plot(x, y)

plt.xlabel("Average Pulse")
plt.ylabel("Calorie Burnage")

plt.show()
```


Create a Title for a Plot

Create a Title for a Plot

```
plt.title("Sports Watch Data", loc='left')
```

Set Font Properties for Title and Labels

```
font1 = {'family':'serif','color':'blue','size':20}
font2 = {'family':'serif','color':'darkred','size':15}

plt.title("Sports Watch Data", fontdict = font1)
plt.xlabel("Average Pulse", fontdict = font2)
plt.ylabel("Calorie Burnage", fontdict = font2)

plt.plot(x, y)
plt.show()
```

Display Multiple Plots

Display Multiple Plots

```
import matplotlib.pyplot as plt
import numpy as np

#plot 1:
x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])

plt.subplot(1, 2, 1)
plt.plot(x,y)

#plot 2:
x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])

plt.subplot(1, 2, 2)
plt.plot(x,y)

plt.show()
```


Display subplots

The subplots() Function

- The subplot() function takes three arguments that describes the layout of the figure.
- The layout is organized in rows and columns, which are represented by the *first* and *second* argument.
- The third argument represents the index of the current plot.

```
plt.subplot(1, 2, 1)
#the figuree has 1 row, 2 columns, and this plot is the first plot.


plt.subplot(1, 2, 2)
#the figure has 1 row, 2 columns, and this plot is the second plot.
```

Try This: If we want a figure with 2 rows an 1 column

Example:

```
import matplotlib.pyplot as plt
import numpy as np

#plot 1:
x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])

plt.subplot(1, 2, 1)
plt.plot(x,y)
plt.title("SALES")

#plot 2:
x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])

plt.subplot(1, 2, 2)
plt.plot(x,y)
plt.title("INCOME")

plt.show()
```

GUI Programming

Most of the programs we have done till now are text-based programming. But many applications need GUI (Graphical User Interface).

Python provides several different options for writing GUI based programs. These are listed below:

- **Tkinter**: It is easiest to start with. Tkinter is Python's standard GUI (graphical user interface) package. It is the most commonly used toolkit for GUI programming in Python.
- **JPython**: It is the Python platform for Java that is providing Python scripts seamless access o Java class Libraries for the local machine.
- **wxPython**: It is an open-source, cross-platform GUI toolkit written in C++. It is one of the alternatives to Tkinter, which is bundled with Python.

There are many other interfaces available for GUI. But these are the most commonly used ones.

Tkinter

It is the standard GUI toolkit for Python. Fredrik Lundh wrote it. For modern Tk binding, Tkinter is implemented as a Python wrapper for the Tcl Interpreter embedded within the interpreter of Python. Tk provides the following widgets:

- Button
- combo-box
- Level
- Entry
- Menu
- menu button
- tk_optoinMenu
- radio button
- Separator
- canvas
- frame
- check-button
- level-frame
- list - box
- message
- progress-bar
- scroll bar
- tree-view, and many more.

Tkinter

Steps to create GUI program using Tkinter:

- Import the module Tkinter
- Build a GUI application (as a window)
- Add those widgets that are discussed above
- Enter the primary, i.e., the main event's loop for taking action when the user triggered the event.

- Importing tkinter is same as importing any other module in the Python code. Note that the name of the module in Python 2.x is 'Tkinter' and in Python 3.x it is 'tkinter'.

Main Methods of GUI Application

There are two main methods used which the user needs to remember while creating the Python application with GUI.

1. **Tk(screenName=None, baseName=None, className='Tk', useTk=1):** To create a main window, tkinter offers a method 'Tk(screenName=None, baseName=None, className='Tk', useTk=1)'. To change the name of the window, you can change the className to the desired one.
2. **mainloop():** There is a method known by the name mainloop() is used when your application is ready to run. mainloop() is an infinite loop used to run the application, wait for an event to occur and process the event as long as the window is not closed.

```
import tkinter
m = tkinter.Tk()
m.mainloop()
```


Geometric Manager Classes

tkinter also offers access to the geometric configuration of the widgets which can organize the widgets in the parent windows. There are mainly three geometry manager classes class.

- **pack() method:**It organizes the widgets in blocks before placing in the parent widget.
- **grid() method:**It organizes the widgets in grid (table-like structure) before placing in the parent widget.
- **place() method:**It organizes the widgets by placing them on specific positions directed by the programmer.

There are a number of widgets which you can put in your tkinter application.

Tkinter grid() Method

This geometry manager organizes widgets in a table-like structure in the parent widget.

```
widget.grid( grid_options )
```

Here is the list of possible options −

- **column** − The column to put widget in; default 0 (leftmost column).
- **columnspan** − How many columns widgetoccupies; default 1.
- **ipadx, ipady** − How many pixels to pad widget, horizontally and vertically, inside widget's borders.
- **padx, pady** − How many pixels to pad widget, horizontally and vertically, outside v's borders.
- **row** − The row to put widget in; default the first row that is still empty.
- **rowspan** − How many rowswidget occupies; default 1.
- **sticky** − What to do if the cell is larger than widget. By default, with sticky='', widget is centered in its cell. sticky may be the string concatenation of zero or more of N, E, S, W, NE, NW, SE, and SW, compass directions indicating the sides and corners of the cell to which widget sticks.

Button

To add a button in your application, this widget is used.
The general syntax is:

```
w=Button(master, option=value)
```

master is the parameter used to represent the parent window.
There are number of options which are used to change the format of the Buttons. Number of options can be passed as parameters separated by commas. Some of them are listed below.

- **activebackground**: to set the background color when button is under the cursor.
- **activeforeground**: to set the foreground color when button is under the cursor.
- **bg**: to set he normal background color.
- **command**: to call a function.
- **font**: to set the font on the button label.
- **image**: to set the image on the button.
- **width**: to set the width of the button.
- **height**: to set the height of the button.

Button (Example)

```
import tkinter as tk
r = tk.Tk()
r.title('Counting Seconds')
button = tk.Button(r, text='Stop', width=25, command=r.destroy)
button.pack()
r.mainloop()
```


Canvas

It is used to draw pictures and other complex layout like graphics, text and widgets. The general syntax is:

```
w = Canvas(master, option=value)
```

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

- **bd**: to set the border width in pixels.
- **bg**: to set the normal background color.
- **cursor**: to set the cursor used in the canvas.
- **highlightcolor**: to set the color shown in the focus highlight.
- **width**: to set the width of the widget.
- **height**: to set the height of the widget.

Canvas (Example)

```
import tkinter as tk
master = tk.Tk()
w = tk.Canvas(master, width=40, height=60)
w.pack()
canvas_height=20
canvas_width=200
y = int(canvas_height / 2)
w.create_line(0, y, canvas_width, y )
w.mainloop()
```


CheckButton

To select any number of options by displaying a number of options to a user as toggle buttons.

```
import tkinter
from tkinter import *
top = tkinter.Tk()
CheckVar1 = IntVar()
CheckVar2 = IntVar()
tkinter.Checkbutton(top, text = "Machine Learning",variable =
CheckVar1,onvalue = 1, offvalue=0).grid(row=0,sticky=W)
tkinter.Checkbutton(top, text = "Deep Learning", variable =
CheckVar2, onvalue = 0, offvalue =1).grid(row=1,sticky=W)
top.mainloop()
```


Entry

Entry: It is used to input the single line text entry from the user.. For multi-line text input, Text widget is used.

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

- **bd**: to set the border width in pixels.
- **bg**: to set the normal background color.
- **cursor**: to set the cursor used.
- **command**: to call a function.
- **highlightcolor**: to set the color shown in the focus highlight.
- **width**: to set the width of the button.
- **height**: to set the height of the button.

Entry (Example)

```
from tkinter import *
master = Tk()
Label(master, text='First Name').grid(row=0)
Label(master, text='Last Name').grid(row=1)
e1 = Entry(master)
e2 = Entry(master)
e1.grid(row=0, column=1)
e2.grid(row=1, column=1)
mainloop()
```


Frame

Frame: It acts as a container to hold the widgets. It is used for grouping and organizing the widgets.

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

- **highlightcolor**: To set the color of the focus highlight when widget has to be focused.
- **bd**: to set the border width in pixels.
- **bg**: to set the normal background color.
- **cursor**: to set the cursor used.
- **width**: to set the width of the widget.
- **height**: to set the height of the widget.

Frame(Example)

```
from tkinter import *
root = Tk()
frame = Frame(root)
frame.pack()
bottomframe = Frame(root)
bottomframe.pack( side = BOTTOM )
redbutton = Button(frame, text = 'Red', fg ='red')
redbutton.pack( side = LEFT)
greenbutton = Button(frame, text = 'Brown', fg='brown')
greenbutton.pack( side = LEFT )
bluebutton = Button(frame, text ='Blue', fg ='blue')
bluebutton.pack( side = LEFT )
blackbutton = Button(bottomframe, text ='Black', fg ='black')
blackbutton.pack( side = BOTTOM)
root.mainloop()
```

Label

Label: It refers to the display box where you can put any text or image which can be updated any time as per the code.

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

- **bg**: to set he normal background color.
- **bg** to set he normal background color.
- **command**: to call a function.
- **font**: to set the font on the button label.
- **image**: to set the image on the button.
- **width**: to set the width of the button.
- **height**" to set the height of the button.

```
w = Label(root, text='Python Programming')
w.pack()
```


ListBox

Listbox: It offers a list to the user from which the user can accept any number of options.

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas.

- **highlightcolor**: To set the color of the focus highlight when widget has to be focused.

```
Lb = Listbox(root)
Lb.insert(1, 'Python')
Lb.insert(2, 'Java')
Lb.insert(3, 'C++')
Lb.insert(4, 'Any other')
Lb.pack()
```


MenuButton

It is a part of top-down menu which stays on the window all the time. Every menubutton has its own functionality.

```
from tkinter import *
import tkinter

top = Tk()
mb =  Menubutton ( top, text = "Menu Items", relief= RAISED)
mb.grid()
mb.menu  =  Menu ( mb, tearoff = 0 )
mb["menu"]  =  mb.menu
cVar  = IntVar()
aVar = IntVar()
mb.menu.add_checkbutton ( label ='Contact', variable = cVar )
mb.menu.add_checkbutton ( label = 'About', variable = aVar )
mb.pack()
top.mainloop()
```

Menu

Menu: It is used to create all kinds of menus used by the application

```
from tkinter import *

root = Tk()
menu = Menu(root)
root.config(menu=menu)
filemenu = Menu(menu)
menu.add_cascade(label='File', menu=filemenu)
filemenu.add_command(label='New')
filemenu.add_command(label='Open...')
filemenu.add_separator()
filemenu.add_command(label='Exit', command=root.quit)
helpmenu = Menu(menu)
menu.add_cascade(label='Help', menu=helpmenu)
helpmenu.add_command(label='About')
mainloop()
```


Message

Message: It refers to the multi-line and non-editable text.

The message widget is similar in its functionality to the Label widget, but it is **more flexible in displaying text**, e.g. the font can be changed while the Label widget can only display text in a single font. It provides a multiline object, that is the text may span more than one line

```
from tkinter import *
main = Tk()
ourMessage ='This is our Message'
messageVar = Message(main, text = ourMessage)
messageVar.config(bg='lightgreen')
messageVar.pack( )
main.mainloop( )
```


RadioButtons

The **Radiobutton** is a standard Tkinter widget used to implement one-of-many selections. **Radiobuttons** can contain text or images, and you can associate a Python function or method with each button. When the button is pressed, Tkinter automatically calls that function or method.

General Syntax:

button = Radiobutton(master, text="Name on Button", variable = "shared variable", value = "values of each button", options = values, …)
shared variable *= A Tkinter variable shared among all Radio buttons*
value *= each radiobutton should have different value otherwise more than 1 radiobutton will get selected.*

RadioButtons (Example)

```
from tkinter import *

def sel():
  selection = "You selected the option " + str(var.get())
  label.config(text = selection)

root = Tk()
var = IntVar()
R1 = Radiobutton(root, text="Option 1", variable=var, value=1, command=sel)
R1.pack( anchor = W )

R2 = Radiobutton(root, text="Option 2", variable=var, value=2, command=sel)
R2.pack( anchor = W )

R3 = Radiobutton(root, text="Option 3", variable=var, value=3, command=sel)
R3.pack( anchor = W)
label = Label(root)
label.pack()
root.mainloop()
```


Database Access

- Python can be used in database applications.
- One of the most popular databases is MySQL.
- You can download a free MySQL database at https://www.mysql.com/downloads/.
- Python needs a MySQL driver to access the MySQL database.
- use PIP to install "MySQL Connector".
- Use command python -m pip install mysql-connector-python

Create Connection

- We can start by creating a connection to the database.
- Use the username and password from your MySQL database:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  password="yourpassword"
)

print(mydb)
```

Creating a Database

To create a database in MySQL, use the "CREATE DATABASE" statement:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  password="yourpassword"
)

mycursor = mydb.cursor()

mycursor.execute("CREATE DATABASE mydatabase")
```


Creating a Table

- To create a table in MySQL, use the "CREATE TABLE" statement.
- Make sure you define the name of the database when you create the connection

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  password="yourpassword",
  database="mydatabase"
)

mycursor = mydb.cursor()

mycursor.execute("CREATE TABLE customers (id INT AUTO_INCREMENT PRIMARY KEY, name
VARCHAR(255), address VARCHAR(255))")
```


Insert Into Table

To fill a table in MySQL, use the "INSERT INTO" statement.

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  password="yourpassword",
  database="mydatabase"
)

mycursor = mydb.cursor()

sql = "INSERT INTO customers (name, address) VALUES (%s, %s)"
val = ("John", "Highway 21")
mycursor.execute(sql, val)

mydb.commit()

print(mycursor.rowcount, "record inserted.")
```

Insert Multiple Rows

- To insert multiple rows into a table, use the executemany() method.
- The second parameter of the executemany() method is a list of tuples, containing the data you want to insert:

```
sql = "INSERT INTO customers (name, address) VALUES (%s, %s)"
val = [
  ('Peter', 'Lowstreet 4'),
  ('Amy', 'Apple st 652'),
  ('Hannah', 'Mountain 21'),
  ('Michael', 'Valley 345'),
  ('Sandy', 'Ocean blvd 2'),
  ('Betty', 'Green Grass 1'),
  ('Richard', 'Sky st 331'),
  ('Susan', 'One way 98'),
  ('Vicky', 'Yellow Garden 2'),
  ('Ben', 'Park Lane 38'),
  ('William', 'Central st 954'),
  ('Chuck', 'Main Road 989'),
  ('Viola', 'Sideway 1633')
]

mycursor.executemany(sql, val)
```


Select From a Table

- To select from a table in MySQL, use the "SELECT" statement:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  password="yourpassword",
  database="mydatabase"
)

mycursor = mydb.cursor()

mycursor.execute("SELECT * FROM customers")

myresult = mycursor.fetchall()

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


Selecting Columns

```
mycursor = mydb.cursor()

mycursor.execute("SELECT name, address FROM customers")

myresult = mycursor.fetchall()

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

Select With a Filter

When selecting records from a table, you can filter the selection by using the "WHERE" statement:

```
mycursor = mydb.cursor()

sql = "SELECT * FROM customers WHERE address ='Park Lane 38'"

mycursor.execute(sql)

myresult = mycursor.fetchall()

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


Prevent SQL Injection

- When query values are provided by the user, you should escape the values.
- This is to prevent SQL injections, which is a common web hacking technique to destroy or misuse your database.
- The mysql.connector module has methods to escape query values:

```
mycursor = mydb.cursor()

sql = "SELECT * FROM customers WHERE address = %s"
adr = ("Yellow Garden 2", )

mycursor.execute(sql, adr)

myresult = mycursor.fetchall()

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


Delete Record

You can delete records from an existing table by using the "DELETE FROM" statement:

```
mycursor = mydb.cursor()

sql = "DELETE FROM customers WHERE address = 'Mountain 21'"

mycursor.execute(sql)

mydb.commit()

print(mycursor.rowcount, "record(s) deleted")
```

Update Records in a Table

You can update existing records in a table by using the "UPDATE" statement:

```
mycursor = mydb.cursor()

sql = "UPDATE customers SET address = 'Canyon 123' WHERE address = 'Valley 345'"

mycursor.execute(sql)

mydb.commit()

print(mycursor.rowcount, "record(s) affected")
```


Limit the Result

You can limit the number of records returned from the query, by using the "LIMIT" statement:

```
mycursor = mydb.cursor()
mycursor.execute("SELECT * FROM customers LIMIT 5")
myresult = mycursor.fetchall()
for x in myresult:
  print(x)
```

Start From Another Position

- If you want to return five records, starting from the third record, you can use the "OFFSET" keyword:

```
mycursor = mydb.cursor()
mycursor.execute("SELECT * FROM customers LIMIT 5 OFFSET 2")
myresult = mycursor.fetchall()
for x in myresult:
  print(x)
```


Drop a Table

You can delete an existing table by using the "DROP TABLE" statement:

```
mycursor = mydb.cursor()
sql = "DROP TABLE customers"
mycursor.execute(sql)
```

Drop Only if Exist

- If the the table you want to delete is already deleted, or for any other reason does not exist, you can use the IF EXISTS keyword to avoid getting an error.

```
mycursor = mydb.cursor()
sql = "DROP TABLE IF EXISTS customers"
mycursor.execute(sql)
```

Join Two or More Tables

- You can combine rows from two or more tables, based on a related column between them, by using a JOIN statement.

Suppose we have a "users" table and a "products" table:

Example

- Join users and products to see the name of the users favorite product:

```
mycursor = mydb.cursor()
sql = "SELECT \
  users.name AS user, \
  products.name AS favorite \
  FROM users \
  INNER JOIN products ON users.fav = products.id"
mycursor.execute(sql)
myresult = mycursor.fetchall()
for x in myresult:
  print(x)
```


LEFT JOIN

- In the example above, Hannah, and Michael were excluded from the result, that is because INNER JOIN only shows the records where there is a match.
- If you want to show all users, even if they do not have a favorite product, use the LEFT JOIN statement:

Example

- Select all users and their favorite product:

```
sql = "SELECT \
  users.name AS user, \
  products.name AS favorite \
  FROM users \
  LEFT JOIN products ON users.fav = products.id"
```


RIGHT JOIN

If you want to return all products, and the users who have them as their favorite, even if no user have them as their favorite, use the RIGHT JOIN statement:

Example

- Select all products, and the user(s) who have them as their favorite:

```
sql = "SELECT \
  users.name AS user, \
  products.name AS favorite \
  FROM users \
  RIGHT JOIN products ON users.fav = products.id"
```