Lab 01
Python Introduction

Objective:

This lab is an introductory session on Python. The lab will equip students with necessary concepts needed to build algorithms in Python.

Activity Outcomes:

The lab will teach students to:

- Basic Operations on strings and numbers
- Basic use of conditionals
- Basic use of loops

Instructor Note:

As a pre-lab activity, read Chapters 3-5 from the book (Introduction to Programming Using Python - Y. Liang (Pearson, 2013)) to gain an insight about python programming and its fundamentals.

1) Useful Concepts

Python refers to the Python programming language (with syntax rules for writing what is considered valid Python code) and the Python interpreter software that reads source code (written in the Python language) and performs its instructions.
Python is a general-purpose programming language. That means you can use Python to write code for any programming task. Python is now used in the Google search engine, in mission-critical projects at NASA, and in transaction processing at the New York Stock Exchange.
Python's most obvious feature is that it uses indentation as a control structure. Indentation is used in Python to delimit blocks. **The number of spaces** is variable, but all statements within the same block must be indented the same amount. The header line for compound statements, such as if, while, def, and class should be terminated with a colon (**:**)

Example: Indentation as a control-structure

```
for i in range(20):
        if i%3 == 0:
                print i
        if i%5 == 0:
                print "Bingo!"
        print "---"
```

Variables
Python is dynamically typed. You do not need to declare variables! The declaration happens automatically when you assign a value to a variable. Variables can change type, simply by assigning them a new value of a different type. Python allows you to assign a single value to several variables simultaneously. You can also assign multiple objects to multiple variables.

Data types in Python
Every value in Python has a datatype. Since everything is an object in Python programming, data types are actually classes and variables are instance (object) of these classes.
There are various data types in Python. Some of the important types are listed below.

Integers
In Python 3, there is effectively no limit to how long an integer value can be. Of course, it is constrained by the amount of memory your system has, as are all things, but beyond that an integer can be as long as you need it to be:

```
>>> print(123123123123123123123123123123123123123123123123 + 1)
123123123123123123123123123123123123123123123123124
```

Python interprets a sequence of decimal digits without any prefix to be a decimal number:

```
>>> print(10)
10
```

The following strings can be prepended to an integer value to indicate a base other than 10:

Prefix	Interpretation	Base
0b (zero + lowercase letter 'b') 0B (zero + uppercase letter 'B')	Binary	2
0o (zero + lowercase letter 'o') 0O (zero + uppercase letter 'O')	Octal	8
0x (zero + lowercase letter 'x') 0X (zero + uppercase letter 'X')	Hexadecimal	16

For example:

```
>>> print(0o10)
8
>>> print(0x10)
16
>>> print(0b10)
2
```

The underlying type of a Python integer, irrespective of the base used to specify it, is called int:

```
>>> type(10)
<class 'int'>
>>> type(0o10)
<class 'int'>
>>> type(0x10)
<class 'int'>
```

Floating-Point Numbers

The float type in Python designates a floating-point number. float values are specified with a decimal point. Optionally, the character e or E followed by a positive or negative integer may be appended to specify scientific notation:

```
>>> 4.2
4.2
>>> type(4.2)
<class 'float'>
>>> 4.
4.0
>>> .2
0.2

>>> .4e7
4000000.0
>>> type(.4e7)
<class 'float'>
>>> 4.2e-4
0.00042
```

Floating-Point Representation

The following is a bit more in-depth information on how Python represents floating-point numbers internally. You can readily use floating-point numbers in Python without understanding them to this level, so don't worry if this seems overly complicated. The information is presented here in case you are curious.

Almost all platforms represent Python float values as 64-bit "double-precision" values, according to the IEEE 754 standard. In that case, the maximum value a floating-point number can have is approximately 1.8×10^{308}. Python will indicate a number greater than that by the string inf:

```
>>> 1.79e308
1.79e+308
>>> 1.8e308
Inf
```

The closest a nonzero number can be to zero is approximately 5.0×10^{-324}. Anything closer to zero than that is effectively zero:

```
>>> 5e-324
5e-324
>>> 1e-325
0.0
```

Floating point numbers are represented internally as binary (base-2) fractions. Most decimal fractions cannot be represented exactly as binary fractions, so in most cases the internal representation of a floating-point number is an approximation of the actual value. In practice, the difference between the actual value and the represented value is very small and should not usually cause significant problems.

Complex Numbers

Complex numbers are specified as <real part>+<imaginary part>j. For example:

```
>>> 2+3j
(2+3j)
>>> type(2+3j)
<class 'complex'>
```

Strings

Strings are sequences of character data. The string type in Python is called str.

String literals may be delimited using either single or double quotes. All the characters between the opening delimiter and matching closing delimiter are part of the string:

```
>>> print("I am a string.")
I am a string.
>>> type("I am a string.")
<class 'str'>

>>> print('I am too.')
I am too.
>>> type('I am too.')
<class 'str'>
```

A string in Python can contain as many characters as you wish. The only limit is your machine's memory resources. A string can also be empty:

```
>>> ''
''
```

What if you want to include a quote character as part of the string itself? Your first impulse might be to try something like this:

```
>>> print('This string contains a single quote (') character.')
SyntaxError: invalid syntax
```

As you can see, that doesn't work so well. The string in this example opens with a single quote, so Python assumes the next single quote, the one in parentheses which was intended to be part of the string, is the closing delimiter. The final single quote is then a stray and causes the syntax error shown.
If you want to include either type of quote character within the string, the simplest way is to delimit the string with the other type. If a string is to contain a single quote, delimit it with double quotes and vice versa:

```
>>> print("This string contains a single quote (') character.")
This string contains a single quote (') character.

>>> print('This string contains a double quote (") character.')
This string contains a double quote (") character.
```

Escape Sequences in Strings

Sometimes, you want Python to interpret a character or sequence of characters within a string differently. This may occur in one of two ways:

- You may want to suppress the special interpretation that certain characters are usually given within a string.
- You may want to apply special interpretation to characters in a string which would normally be taken literally.

You can accomplish this using a backslash (\) character. A backslash character in a string indicates that one or more characters that follow it should be treated specially. (This is referred to as an escape sequence, because the backslash causes the subsequent character sequence to "escape" its usual meaning.)

Suppressing Special Character Meaning

You have already seen the problems you can come up against when you try to include quote characters in a string. If a string is delimited by single quotes, you can't directly specify a single quote character as part of the string because, for that string, the single quote has special meaning—it terminates the string:

```
>>> print('This string contains a single quote (') character.')
SyntaxError: invalid syntax
```

Specifying a backslash in front of the quote character in a string "escapes" it and causes Python to suppress its usual special meaning. It is then interpreted simply as a literal single quote character:

```
>>> print('This string contains a single quote (\') character.')
This string contains a single quote (') character.
```

The same works in a string delimited by double quotes as well:

```
>>> print("This string contains a double quote (\") character.")
This string contains a double quote (") character.
```

The following is a table of escape sequences which cause Python to suppress the usual special interpretation of a character in a string:

Escape Sequence	Usual Interpretation of Character(s) After Backslash	"Escaped" Interpretation
\'	Terminates string with single quote opening delimiter	Literal single quote (') character
\"	Terminates string with double quote opening delimiter	Literal double quote (") character
\<newline>	Terminates input line	Newline is ignored
\\	Introduces escape sequence	Literal backslash (\) character

Ordinarily, a newline character terminates line input. So pressing Enter in the middle of a string will cause Python to think it is incomplete:

```
>>> print('a

SyntaxError: EOL while scanning string literal
```

To break up a string over more than one line, include a backslash before each newline, and the newlines will be ignored:

```
>>> print('a\
... b\
... c')
Abc
```

To include a literal backslash in a string, escape it with a backslash:

```
>>> print('foo\\bar')
foo\bar
```

Applying Special Meaning to Characters

Next, suppose you need to create a string that contains a tab character in it. Some text editors may allow you to insert a tab character directly into your code. But many programmers consider that poor practice, for several reasons:

- The computer can distinguish between a tab character and a sequence of space characters, but you can't. To a human reading the code, tab and space characters are visually indistinguishable.
- Some text editors are configured to automatically eliminate tab characters by expanding them to the appropriate number of spaces.
- Some Python REPL environments will not insert tabs into code.

In Python (and almost all other common computer languages), a tab character can be specified by the escape sequence \t:

```
>>> print('foo\tbar')
foo	bar
```

The escape sequence \t causes the t character to lose its usual meaning, that of a literal t. Instead, the combination is interpreted as a tab character.

Here is a list of escape sequences that cause Python to apply special meaning instead of interpreting literally:

Escape Sequence	"Escaped" Interpretation
\a	ASCII Bell (BEL) character
\b	ASCII Backspace (BS) character
\f	ASCII Formfeed (FF) character
\n	ASCII Linefeed (LF) character
\N{<name>}	Character from Unicode database with given <name>
\r	ASCII Carriage Return (CR) character
\t	ASCII Horizontal Tab (TAB) character
\uxxxx	Unicode character with 16-bit hex value xxxx
\Uxxxxxxxx	Unicode character with 32-bit hex value xxxxxxxx
\v	ASCII Vertical Tab (VT) character
\ooo	Character with octal value ooo
\xhh	Character with hex value hh

Examples:

```
>>> print("a\tb")
a	b
>>> print("a\141\x61")
aaa
>>> print("a\nb")
a
b
>>> print('\u2192 \N{rightwards arrow}')
→ →
```

This type of escape sequence is typically used to insert characters that are not readily generated from the keyboard or are not easily readable or printable.

Raw Strings

A raw string literal is preceded by r or R, which specifies that escape sequences in the associated string are not translated. The backslash character is left in the string:

```
>>> print('foo\nbar')
```

```
foo
bar
>>> print(r'foo\nbar')
foo\nbar

>>> print('foo\\bar')
foo\bar
>>> print(R'foo\\bar')
foo\\bar
```

Triple-Quoted Strings

There is yet another way of delimiting strings in Python. Triple-quoted strings are delimited by matching groups of three single quotes or three double quotes. Escape sequences still work in triple-quoted strings, but single quotes, double quotes, and newlines can be included without escaping them. This provides a convenient way to create a string with both single and double quotes in it:

```
>>> print('''This string has a single (') and a double (") quote.''')
This string has a single (') and a double (") quote.
```

Because newlines can be included without escaping them, this also allows for multiline strings:

```
>>> print("""This is a
string that spans
across several lines""")
This is a
string that spans
across several lines
```

You will see in the upcoming tutorial on Python Program Structure how triple-quoted strings can be used to add an explanatory comment to Python code.

Boolean Type, Boolean Context, and "Truthiness"

Python 3 provides a Boolean data type. Objects of Boolean type may have one of two values, True or False:

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

As you will see in upcoming tutorials, expressions in Python are often evaluated in Boolean context, meaning they are interpreted to represent truth or falsehood. A value that is true in Boolean context is sometimes said to be "truthy," and one that is false in Boolean context is said to be "falsy." (You may also see "falsy" spelled "falsey.")

The "truthiness" of an object of Boolean type is self-evident: Boolean objects that are equal to True are truthy (true), and those equal to False are falsy (false). But non-Boolean objects can be evaluated in Boolean context as well and determined to be true or false.

2) Solved Lab Activites

Sr.No	*Allocated Time*	*Level of Complexity*	*CLO Mapping*
1	*5*	*Low*	*CLO-6*
2	*5*	*Low*	*CLO-6*
3	*10*	*Medium*	*CLO-6*
4	*5*	*Low*	*CLO-6*
5	*5*	*Low*	*CLO-6*
6	*5*	*Low*	*CLO-6*
7	*10*	*Medium*	*CLO-6*

Activity 1:

Let us take an integer from user as input and check whether the given value is even or not. If the given value is not even then it means that it will be odd. So here we need to use if-else statement an demonstrated below

A. Create a new Python file from Python Shell and type the following code.
B. Run the code by pressing F5.

```
n=input("Enter a number ")
if int(n)%2==0:
    print("The given number is an even number")
else:
    print("The given number is an odd number")
```

Output

```
Enter a number 11
The given number is an odd number
>>>
```

Activity 2:

Write a Python code to keep accepting integer values from user until 0 is entered. Display sum of the given values.

Solution:

```
sum=0
s=input("Enter an integer value...")
n=int(s)
while n!=0:
    sum=sum+n
    s=input("Enter an integer value...")
    n=int(s)
print("Sum of given values is ",sum)
```

Output

```
Enter an integer value...10
Enter an integer value...521
Enter an integer value...5
Enter an integer value...22
Enter an integer value...0
Sum of given values is  558
>>>
```

Activity 3:

Write a Python code to accept an integer value from user and check that whether the given value is prime number or not.

Solution:

```
isPrime = True
i=2
n=int(input("enter a number"))
while i<n:
    remainder=n%i
    if remainder==0:
        isPrime=False
        break
    else:
        i=i+1

if isPrime:
    print("Number is Prime")
else:
    print("Number is not Prime")
```

Activity 4:

Accept 5 integer values from user and display their sum. Draw flowchart before coding in python.

Solution:

Create a new Python file from Python Shell and type the following code. Run the code by pressing F5.

```
summ = 0
i=0
while i<=4:
    s=input("enter a number")
    n=int(s)
    summ=summ+n
    i=i+1

print("sum is ",summ)
```

You will get the following output.

```
enter a number1
enter a number2
enter a number3
enter a number4
enter a number5
sum is  15
>>>
```

Activity 5:

Calculate the sum of all the values between 0-10 using while loop.

Solution:

Create a new Python file from Python Shell and type the following code.
Run the code by pressing F5.

```
summation = 0
i=1
while i<=10:
    summation=summation+i
    i=i+1

print("sum is ",summation)
```

You will get the following output.

```
sum is  55
>>>
```

Activity 6:

Take input from the keyboard and use it in your program.

Solution:

In Python and many other programming languages you can get user input. In Python the input() function will ask keyboard input from the user.The input function prompts text if a parameter is given. The function reads input from the keyboard, converts it to a string and removes the newline (Enter).Type and experiment with the script below.

```
#!/usr/bin/env python3

name = input('What is your name? ')
print('Hello ' + name)

job = input('What is your job? ')
print('Your job is ' + job)

num = input("Give me a number? ")
print('You said: ' + str(num))
```

Activity 7:

Generate a random number between 1 and 9 (including 1 and 9). Ask the user to guess the number, then tell them whether they guessed too low, too high, or exactly right. (Hint: remember to use the user input lessons from the very first exercise)

Extras:

Keep the game going until the user types "exit"

Keep track of how many guesses the user has taken, and when the game ends, print this out.

Solution:

```
import random
# Awroken

MINIMUM = 1
MAXIMUM = 9
NUMBER = random.randint(MINIMUM, MAXIMUM)
GUESS = None
ANOTHER = None
TRY = 0
RUNNING = True

print "Alright..."

while RUNNING:
   GUESS = raw_input("What is your lucky number? ")
   if int(GUESS) < NUMBER:
      print "Wrong, too low."
   elif int(GUESS) > NUMBER:
      print "Wrong, too high."
   elif GUESS.lower() == "exit":
      print "Better luck next time."
   elif int(GUESS) == NUMBER:
      print "Yes, that's the one, %s." % str(NUMBER)
      if TRY < 2:
         print "Impressive, only %s tries." % str(TRY)
      elif TRY > 2 and TRY < 10:
         print "Pretty good, %s tries." % str(TRY)
      else:
         print "Bad, %s tries." % str(TRY)
      RUNNING = False
   TRY += 1
```

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficult and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab.

Lab Task 1:

Write a program that prompts the user to input an integer and then outputs the number with the digits reversed. For example, if the input is 12345, the output should be 54321.

Lab Task 2:

Write a program that reads a set of integers, and then prints the sum of the even and odd integers.

Lab Task 3:

Fibonacci series is that when you add the previous two numbers the next number is formed. You have to start from 0 and 1.
E.g. 0+1=1 → 1+1=2 → 1+2=3 → 2+3=5 → 3+5=8 → 5+8=13

So the series becomes
0 1 1 2 3 5 8 13 21 34 55 ……

Steps: You have to take an input number that shows how many terms to be displayed. Then use loops for displaying the Fibonacci series up to that term e.g. input no is =6 the output should be
0 1 1 2 3 5

Lab Task 4:

Write a Python code to accept marks of a student from 1-100 and display the grade according to the following formula.

Grade F if marks are less than 50
Grade E if marks are between 50 to 60
Grade D if marks are between 61 to 70
Grade C if marks are between 71 to 80
Grade B if marks are between 81 to 90
Grade A if marks are between 91 to 100

Lab Task 5:

Write a program that takes a number from user and calculate the factorial of that number.

Lab 02
Python Lists and Dictionaries

Objective:

This lab will give you practical implementation of different types of **sequences** including **lists, tuples, sets and dictionaries**. We will use lists alongside loops in order to know about indexing individual items of these containers. This lab will also allow students to write their own **functions.**

Activity Outcomes:

This lab teaches you the following topics:

- How to use lists, tuples, sets and dictionaries How to use loops with lists
- How to write customized functions

Instructor Note:

As a pre-lab activity, read Chapters 6, 10 and 14 from the book (*Introduction to Programming Using Python - Y. Liang (Pearson, 2013)*) to gain an insight about python programming and its fundamentals.

1) Useful Concepts

Python provides different types of data structures as sequences. In a sequence, there are more than one values and each value has its own index. The first value will have an index 0 in python, the second value will have index 1 and so on. These indices are used to access a particular value in the sequence.

Python Lists:

Lists are just like dynamically sized arrays, declared in other languages (vector in C++ and ArrayList in Java). Lists need not be homogeneous always which makes it the most powerful tool in Python. A single list may contain DataTypes like Integers, Strings, as well as Objects. Lists are mutable, and hence, they can be altered even after their creation.

List in Python are ordered and have a definite count. The elements in a list are indexed according to a definite sequence and the indexing of a list is done with 0 being the first index. Each element in the list has its definite place in the list, which allows duplicating of elements in the list, with each element having its own distinct place and credibility.

Creating a List

Lists in Python can be created by just placing the sequence inside the square brackets[]. Unlike Sets, a list doesn't need a built-in function for the creation of a list.

```
# Python program to demonstrate
# Creation of List

# Creating a List
List = []
print("Blank List: ")
print(List)

# Creating a List of numbers
List = [10, 20, 14]
print("\nList of numbers: ")
print(List)

# Creating a List of strings and accessing
# using index
List = ["Geeks", "For", "Geeks"]
print("\nList Items: ")
print(List[0])
print(List[2])

# Creating a Multi-Dimensional List
# (By Nesting a list inside a List)
List = [['Geeks', 'For'], ['Geeks']]
print("\nMulti-Dimensional List: ")
print(List)
```

Output:
Blank List:
[]

List of numbers:
[10, 20, 14]

List Items
Geeks
Geeks

Multi-Dimensional List:
[['Geeks', 'For'], ['Geeks']]

Creating a list with multiple distinct or duplicate elements

A list may contain duplicate values with their distinct positions and hence, multiple distinct or duplicate values can be passed as a sequence at the time of list creation.

```
# Creating a List with
# the use of Numbers
# (Having duplicate values)
List = [1, 2, 4, 4, 3, 3, 3, 6, 5]
print("\nList with the use of Numbers: ")
print(List)

# Creating a List with
# mixed type of values
# (Having numbers and strings)
List = [1, 2, 'Geeks', 4, 'For', 6, 'Geeks']
print("\nList with the use of Mixed Values: ")
print(List)
```

Output:
List with the use of Numbers:
[1, 2, 4, 4, 3, 3, 3, 6, 5]

List with the use of Mixed Values:
[1, 2, 'Geeks', 4, 'For', 6, 'Geeks']

Knowing the size of List

```
# Creating a List
List1 = []
```

```
print(len(List1))

# Creating a List of numbers
List2 = [10, 20, 14]
print(len(List2))
```

Output:
0
3

Adding Elements to a List
Using append() method
Elements can be added to the List by using the built-in **append()** function. Only one element at a time can be added to the list by using the append() method, for the addition of multiple elements with the append() method, loops are used. Tuples can also be added to the list with the use of the append method because tuples are immutable. Unlike Sets, Lists can also be added to the existing list with the use of the append() method.

```
# Python program to demonstrate
# Addition of elements in a List

# Creating a List
List = []
print("Initial blank List: ")
print(List)

# Addition of Elements
# in the List
List.append(1)
List.append(2)
List.append(4)
print("\nList after Addition of Three elements: ")
print(List)

# Adding elements to the List
# using Iterator
for i in range(1, 4):
    List.append(i)
print("\nList after Addition of elements from 1-3: ")
print(List)

# Adding Tuples to the List
List.append((5, 6))
print("\nList after Addition of a Tuple: ")
```

```
print(List)

# Addition of List to a List
List2 = ['For', 'Geeks']
List.append(List2)
print("\nList after Addition of a List: ")
print(List)
```

Output:

```
Initial blank List:
[]

List after Addition of Three elements:
[1, 2, 4]

List after Addition of elements from 1-3:
[1, 2, 4, 1, 2, 3]

List after Addition of a Tuple:
[1, 2, 4, 1, 2, 3, (5, 6)]

List after Addition of a List:
[1, 2, 4, 1, 2, 3, (5, 6), ['For', 'Geeks']]
```

Using insert() method

append() method only works for the addition of elements at the end of the List, for the addition of elements at the desired position, insert() method is used. Unlike append() which takes only one argument, the insert() method requires two arguments(position, value).

```
# Python program to demonstrate
# Addition of elements in a List

# Creating a List
List = [1,2,3,4]
print("Initial List: ")
print(List)

# Addition of Element at
# specific Position
# (using Insert Method)
List.insert(3, 12)
List.insert(0, 'Geeks')
print("\nList after performing Insert Operation: ")
print(List)
```

Output:
Initial List:
[1, 2, 3, 4]

List after performing Insert Operation:
['Geeks', 1, 2, 3, 12, 4]

Using extend() method

Other than append() and insert() methods, there's one more method for the Addition of elements, **extend()**, this method is used to add multiple elements at the same time at the end of the list.

```
# Python program to demonstrate
# Addition of elements in a List

# Creating a List
List = [1, 2, 3, 4]
print("Initial List: ")
print(List)

# Addition of multiple elements
# to the List at the end
# (using Extend Method)
List.extend([8, 'Geeks', 'Always'])
print("\nList after performing Extend Operation: ")
print(List)
```

Output:
Initial List:
[1, 2, 3, 4]

List after performing Extend Operation:
[1, 2, 3, 4, 8, 'Geeks', 'Always']

Accessing elements from the List

In order to access the list items refer to the index number. Use the index operator [ ] to access an item in a list. The index must be an integer. Nested lists are accessed using nested indexing.

```
# Python program to demonstrate
# accessing of element from list

# Creating a List with
# the use of multiple values
List = ["Geeks", "For", "Geeks"]
```

```
 # accessing a element from the
# list using index number
print("Accessing a element from the list")
print(List[0])
print(List[2])

# Creating a Multi-Dimensional List
# (By Nesting a list inside a List)
List = [['Geeks', 'For'], ['Geeks']]

# accessing an element from the
# Multi-Dimensional List using
# index number
print("Accessing a element from a Multi-Dimensional list")
print(List[0][1])
print(List[1][0])
```

Output:

```
Accessing a element from the list
Geeks
Geeks
Accessing a element from a Multi-Dimensional list
For
Geeks
```

Negative indexing

In Python, negative sequence indexes represent positions from the end of the array. Instead of having to compute the offset as in List[len(List)-3], it is enough to just write List[-3]. Negative indexing means beginning from the end, -1 refers to the last item, -2 refers to the second-last item, etc.

```
List = [1, 2, 'Geeks', 4, 'For', 6, 'Geeks']

# accessing an element using negative indexing
print("Accessing element using negative indexing")

# print the last element of list
print(List[-1])

# print the third last element of list
print(List[-3])
```

Output:

```
Accessing element using negative indexing
Geeks
For
```

Removing Elements from the List

Using remove() method

Elements can be removed from the List by using the built-in **remove()** function but an Error arises if the element doesn't exist in the list. Remove() method only removes one element at a time, to remove a range of elements, the iterator is used. The remove() method removes the specified item.

```
# Python program to demonstrate
# Removal of elements in a List

# Creating a List
List = [1, 2, 3, 4, 5, 6,
       7, 8, 9, 10, 11, 12]
print("Initial List: ")
print(List)

# Removing elements from List
# using Remove() method
List.remove(5)
List.remove(6)
print("\nList after Removal of two elements: ")
print(List)

# Removing elements from List
# using iterator method
for i in range(1, 5):
    List.remove(i)
print("\nList after Removing a range of elements: ")
print(List)
```

Output:

```
Initial List:
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

List after Removal of two elements:
[1, 2, 3, 4, 7, 8, 9, 10, 11, 12]

List after Removing a range of elements:
[7, 8, 9, 10, 11, 12]
```

Using pop() method

Pop() function can also be used to remove and return an element from the list, but by default it removes only the last element of the list, to remove an element from a specific position of the List, the index of the element is passed as an argument to the pop() method.

```
List = [1,2,3,4,5]
 # Removing element from the
```

```
# Set using the pop() method
List.pop()
print("\nList after popping an element: ")
print(List)

# Removing element at a
# specific location from the
# Set using the pop() method
List.pop(2)
print("\nList after popping a specific element: ")
print(List)
```

Output:

List after popping an element:
[1, 2, 3, 4]

List after popping a specific element:
[1, 2, 4]

Slicing of a List

In Python List, there are multiple ways to print the whole List with all the elements, but to print a specific range of elements from the list, we use the Slice operation. Slice operation is performed on Lists with the use of a colon(:). To print elements from beginning to a range use [: Index], to print elements from end-use [:-Index], to print elements from specific Index till the end use [Index:], to print elements within a range, use [Start Index:End Index] and to print the whole List with the use of slicing operation, use [:]. Further, to print the whole List in reverse order, use [::-1].

Note – To print elements of List from rear-end, use Negative Indexes.

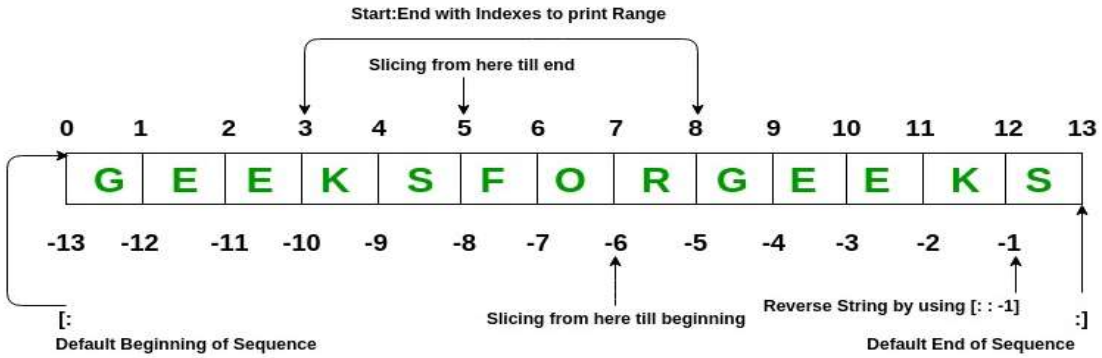

Figure 1 - List

```
# Python program to demonstrate
# Removal of elements in a List

# Creating a List
List = ['G', 'E', 'E', 'K', 'S', 'F',
      'O', 'R', 'G', 'E', 'E', 'K', 'S']
```

```
print("Initial List: ")
print(List)

# Print elements of a range
# using Slice operation
Sliced_List = List[3:8]
print("\nSlicing elements in a range 3-8: ")
print(Sliced_List)

# Print elements from a
# pre-defined point to end
Sliced_List = List[5:]
print("\nElements sliced from 5th "
      "element till the end: ")
print(Sliced_List)

# Printing elements from
# beginning till end
Sliced_List = List[:]
print("\nPrinting all elements using slice operation: ")
print(Sliced_List)
```

Output:

```
Initial List:
['G', 'E', 'E', 'K', 'S', 'F', 'O', 'R', 'G', 'E', 'E', 'K', 'S']

Slicing elements in a range 3-8:
['K', 'S', 'F', 'O', 'R']

Elements sliced from 5th element till the end:
['F', 'O', 'R', 'G', 'E', 'E', 'K', 'S']

Printing all elements using slice operation:
['G', 'E', 'E', 'K', 'S', 'F', 'O', 'R', 'G', 'E', 'E', 'K', 'S']
```

Negative index List slicing

```
# Creating a List
List = ['G', 'E', 'E', 'K', 'S', 'F',
        'O', 'R', 'G', 'E', 'E', 'K', 'S']
print("Initial List: ")
print(List)

# Print elements from beginning
# to a pre-defined point using Slice
```

```
Sliced_List = List[:-6]
print("\nElements sliced till 6th element from last: ")
print(Sliced_List)

# Print elements of a range
# using negative index List slicing
Sliced_List = List[-6:-1]
print("\nElements sliced from index -6 to -1")
print(Sliced_List)

# Printing elements in reverse
# using Slice operation
Sliced_List = List[::-1]
print("\nPrinting List in reverse: ")
print(Sliced_List)
```

Output:

```
Initial List:
['G', 'E', 'E', 'K', 'S', 'F', 'O', 'R', 'G', 'E', 'E', 'K', 'S']

Elements sliced till 6th element from last:
['G', 'E', 'E', 'K', 'S', 'F', 'O']

Elements sliced from index -6 to -1
['R', 'G', 'E', 'E', 'K']

Printing List in reverse:
['S', 'K', 'E', 'E', 'G', 'R', 'O', 'F', 'S', 'K', 'E', 'E', 'G']
```

List Comprehension

List comprehensions are used for creating new lists from other iterables like tuples, strings, arrays, lists, etc.

A list comprehension consists of brackets containing the expression, which is executed for each element along with the for loop to iterate over each element.

Syntax:

newList = [expression(element) for element in oldList if condition]

Example:

```
# Python program to demonstrate list comprehension in Python
 # below list contains square of all odd numbers from range 1 to 10

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

Output:

```
[1, 9, 25, 49, 81]
```

For better understanding, the above code is similar to –

```
# for understanding, above generation is same as,
odd_square = []

for x in range(1, 11):
   if x % 2 == 1:
      odd_square.append(x**2)

print(odd_square)
```

Output:
[1, 9, 25, 49, 81]

Dictionary in Python is an unordered collection of data values, used to store data values like a map, which, unlike other Data Types that hold only a single value as an element, Dictionary holds **key:value** pair. Key-value is provided in the dictionary to make it more optimized.

Note – Keys in a dictionary don't allow Polymorphism.
Disclamer: *It is important to note that Dictionaries have been modified to maintain insertion order with the release of Python 3.7, so they are now ordered collection of data values.*

Creating a Dictionary

In Python, a Dictionary can be created by placing a sequence of elements within curly {} braces, separated by 'comma'. Dictionary holds pairs of values, one being the Key and the other corresponding pair element being its **Key:value**. Values in a dictionary can be of any data type and can be duplicated, whereas keys can't be repeated and must be *immutable*.

```
# Creating a Dictionary
# with Integer Keys
Dict = {1: 'Geeks', 2: 'For', 3: 'Geeks'}
print("\nDictionary with the use of Integer Keys: ")
print(Dict)

# Creating a Dictionary
# with Mixed keys
Dict = {'Name': 'Geeks', 1: [1, 2, 3, 4]}
print("\nDictionary with the use of Mixed Keys: ")
print(Dict)
```

Output:
Dictionary with the use of Integer Keys:
{1: 'Geeks', 2: 'For', 3: 'Geeks'}
Dictionary with the use of Mixed Keys:
{1: [1, 2, 3, 4], 'Name': 'Geeks'}

Dictionary can also be created by the built-in function dict(). An empty dictionary can be created by just placing to curly braces{}.

```
# Creating an empty Dictionary
Dict = {}
print("Empty Dictionary: ")
print(Dict)

# Creating a Dictionary
# with dict() method
Dict = dict({1: 'Geeks', 2: 'For', 3:'Geeks'})
print("\nDictionary with the use of dict(): ")
print(Dict)

# Creating a Dictionary
# with each item as a Pair
Dict = dict([(1, 'Geeks'), (2, 'For')])
print("\nDictionary with each item as a pair: ")
print(Dict)
```

Output:

```
Empty Dictionary:
{}
Dictionary with the use of dict():
{1: 'Geeks', 2: 'For', 3: 'Geeks'}
Dictionary with each item as a pair:
{1: 'Geeks', 2: 'For'}
```

Nested Dictionary:

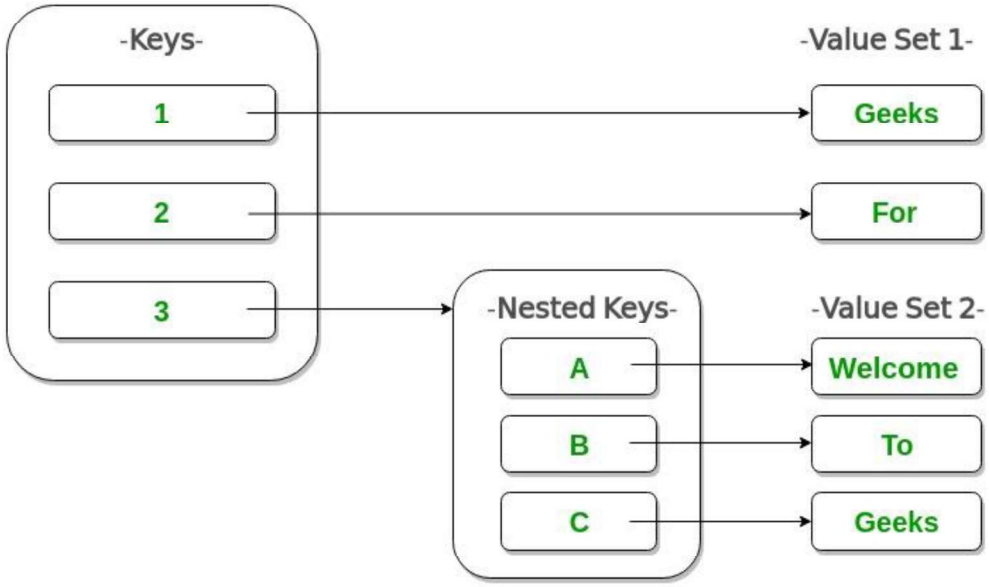

Figure 2 - Dictionary

```
# Creating a Nested Dictionary
# as shown in the below image
Dict = {1: 'Geeks', 2: 'For',
       3:{'A' : 'Welcome', 'B' : 'To', 'C' : 'Geeks'}}

print(Dict)
```

Output:

```
{1: 'Geeks', 2: 'For', 3: {'A': 'Welcome', 'B': 'To', 'C': 'Geeks'}}
```

Adding elements to a Dictionary

In Python Dictionary, the Addition of elements can be done in multiple ways. One value at a time can be added to a Dictionary by defining value along with the key e.g. Dict[Key] = 'Value'. Updating an existing value in a Dictionary can be done by using the built-in **update()** method. Nested key values can also be added to an existing Dictionary.

Note- While adding a value, if the key-value already exists, the value gets updated otherwise a new Key with the value is added to the Dictionary.

```
# Creating an empty Dictionary
Dict = {}
print("Empty Dictionary: ")
print(Dict)

# Adding elements one at a time
Dict[0] = 'Geeks'
Dict[2] = 'For'
Dict[3] = 1
print("\nDictionary after adding 3 elements: ")
print(Dict)

# Adding set of values
# to a single Key
Dict['Value_set'] = 2, 3, 4
print("\nDictionary after adding 3 elements: ")
print(Dict)

# Updating existing Key's Value
Dict[2] = 'Welcome'
print("\nUpdated key value: ")
print(Dict)

# Adding Nested Key value to Dictionary
Dict[5] = {'Nested' :{'1' : 'Life', '2' : 'Geeks'}}
```

```
print("\nAdding a Nested Key: ")
print(Dict)
```

Output:

```
Empty Dictionary:
{}

Dictionary after adding 3 elements:
{0: 'Geeks', 2: 'For', 3: 1}

Dictionary after adding 3 elements:
{0: 'Geeks', 2: 'For', 3: 1, 'Value_set': (2, 3, 4)}

Updated key value:
{0: 'Geeks', 2: 'Welcome', 3: 1, 'Value_set': (2, 3, 4)}

Adding a Nested Key:
{0: 'Geeks', 2: 'Welcome', 3: 1, 5: {'Nested': {'1': 'Life', '2': 'Geeks'}}, 'Value_set': (2, 3, 4)}
```

Accessing elements from a Dictionary

In order to access the items of a dictionary refer to its key name. Key can be used inside square brackets.

```
# Python program to demonstrate
# accessing a element from a Dictionary

# Creating a Dictionary
Dict = {1: 'Geeks', 'name': 'For', 3: 'Geeks'}

# accessing a element using key
print("Accessing a element using key:")
print(Dict['name'])

# accessing a element using key
print("Accessing a element using key:")
print(Dict[1])
```

Output:

```
Accessing a element using key:
For
Accessing a element using key:
Geeks
```

There is also a method called **get()** that will also help in accessing the element from a dictionary.

```
# Creating a Dictionary
Dict = {1: 'Geeks', 'name': 'For', 3: 'Geeks'}

# accessing a element using get()
# method
print("Accessing a element using get:")
print(Dict.get(3))
```

Output:

Accessing a element using get:
Geeks

Accessing an element of a nested dictionary

In order to access the value of any key in the nested dictionary, use indexing [] syntax.

```
# Creating a Dictionary
Dict = {'Dict1': {1: 'Geeks'},
        'Dict2': {'Name': 'For'}}

# Accessing element using key
print(Dict['Dict1'])
print(Dict['Dict1'][1])
print(Dict['Dict2']['Name'])
```

Output:

{1: 'Geeks'}
Geeks
For

Removing Elements from Dictionary

Using del keyword

In Python Dictionary, deletion of keys can be done by using the **del** keyword. Using the del keyword, specific values from a dictionary as well as the whole dictionary can be deleted. Items in a Nested dictionary can also be deleted by using the del keyword and providing a specific nested key and particular key to be deleted from that nested Dictionary.

```
# Initial Dictionary
Dict = { 5 : 'Welcome', 6 : 'To', 7 : 'Geeks',
        'A' : {1 : 'Geeks', 2 : 'For', 3 : 'Geeks'},
        'B' : {1 : 'Geeks', 2 : 'Life'}}
print("Initial Dictionary: ")
print(Dict)

# Deleting a Key value
del Dict[6]
```

```
print("\nDeleting a specific key: ")
print(Dict)

# Deleting a Key from
# Nested Dictionary
del Dict['A'][2]
print("\nDeleting a key from Nested Dictionary: ")
print(Dict)
```

Output:

Initial Dictionary:
{'A': {1: 'Geeks', 2: 'For', 3: 'Geeks'}, 'B': {1: 'Geeks', 2: 'Life'}, 5: 'Welcome', 6: 'To', 7: 'Geeks'}

Deleting a specific key:
{'A': {1: 'Geeks', 2: 'For', 3: 'Geeks'}, 'B': {1: 'Geeks', 2: 'Life'}, 5: 'Welcome', 7: 'Geeks'}

Deleting a key from Nested Dictionary:
{'A': {1: 'Geeks', 3: 'Geeks'}, 'B': {1: 'Geeks', 2: 'Life'}, 5: 'Welcome', 7: 'Geeks'}

Using pop() method

Pop() method is used to return and delete the value of the key specified.

```
# Creating a Dictionary
Dict = {1: 'Geeks', 'name': 'For', 3: 'Geeks'}

# Deleting a key
# using pop() method
pop_ele = Dict.pop(1)
print('\nDictionary after deletion: ' + str(Dict))
print('Value associated to poped key is: ' + str(pop_ele))
```

Output:

Dictionary after deletion: {3: 'Geeks', 'name': 'For'}
Value associated to poped key is: Geeks

Using popitem() method

The popitem() returns and removes an arbitrary element (key, value) pair from the dictionary.

```
# Creating Dictionary
Dict = {1: 'Geeks', 'name': 'For', 3: 'Geeks'}

# Deleting an arbitrary key
# using popitem() function
pop_ele = Dict.popitem()
print("\nDictionary after deletion: " + str(Dict))
```

```
print("The arbitrary pair returned is: " + str(pop_ele))
```

Output:

```
Dictionary after deletion: {3: 'Geeks', 'name': 'For'}
The arbitrary pair returned is: (1, 'Geeks')
```

Using clear() method

All the items from a dictionary can be deleted at once by using **clear()** method.

```
# Creating a Dictionary
Dict = {1: 'Geeks', 'name': 'For', 3: 'Geeks'}


# Deleting entire Dictionary
Dict.clear()
print("\nDeleting Entire Dictionary: ")
print(Dict)
```

Output:

```
Deleting Entire Dictionary:
{}
```

Dictionary Methods

Methods	Description
copy()	They copy() method returns a shallow copy of the dictionary.
clear()	The clear() method removes all items from the dictionary.
pop()	Removes and returns an element from a dictionary having the given key.
popitem()	Removes the arbitrary key-value pair from the dictionary and returns it as tuple.
get()	It is a conventional method to access a value for a key.
dictionary_name.values()	returns a list of all the values available in a given dictionary.
str()	Produces a printable string representation of a dictionary.
update()	Adds dictionary dict2's key-values pairs to dict
setdefault()	Set dict[key]=default if key is not already in dict
keys()	Returns list of dictionary dict's keys
items()	Returns a list of dict's (key, value) tuple pairs
has_key()	Returns true if key in dictionary dict, false otherwise
fromkeys()	Create a new dictionary with keys from seq and values set to value.
type()	Returns the type of the passed variable.
cmp()	Compares elements of both dict.

2) Solved Lab Activites:

Sr.No	*Allocated Time*	*Level of Complexity*	*CLO Mapping*
1	*10*	*Low*	*CLO-6*
2	*10*	*Low*	*CLO-6*
3	*10*	*Low*	*CLO-6*
4	*15*	*Medium*	*CLO-6*
5	*15*	*Medium*	*CLO-6*
6	*15*	*Medium*	*CLO-6*

Activity 1

Accept two lists from user and display their join.

Solution:

```
myList1=[]
print("Enter objects of first list...")
for i in range(5):
    val=input("Enter a value:")
    n=int(val)
    myList1.append(n)

myList2=[]
print("Enter objects of second list...")
for i in range(5):
    val=input("Enter a value:")
    n=int(val)
    myList2.append(n)

list3=myList1+myList2;
print(list3)
```

You will get the following output.

```
Enter objects of first list...
Enter a value:1
Enter a value:2
Enter a value:3
Enter a value:4
Enter a value:5
Enter objects of second list...
Enter a value:6
Enter a value:7
Enter a value:8
Enter a value:9
Enter a value:0
[1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
>>>
```

Activity 2:

A palindrome is a string which is same read forward or backwards.
For example: "dad" is the same in forward or reverse direction. Another example is "aibohphobia" which literally means, an irritable fear of palindromes.
Write a function in python that receives a string and returns True if that string is a palindrome and False otherwise. Remember that difference between upper and lower case characters are ignored during this determination.

Solution:

```
def isPalindrome(word):
    temp=word[::-1]
    if temp.capitalize()==word.capitalize():
        return True
    else:
        return False


print(isPalindrome("deed"))
```

Activity 3:

Imagine two matrices given in the form of 2D lists as under; a = [[1, 0, 0], [0, 1, 0], [0, 0, 1]]
b = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

Write a python code that finds another matrix/2D list that is a product of and b, i.e., C=a*b
Solution:

```
for indrow in range (3):
    c.append ([])
    for indcol in range(3):
        c[indrow].append (0)
        for indaux in range (3):
            c[indrow][indcol] += a[indrow][indaux] * b[indcol][inc

print (c)
```

Activity 4:

A closed polygon with N sides can be represented as a list of tuples of N connected coordinates, i.e., [(x_1,y_1), (x_2,y_2), (x_3,y_3), . . . , (x_N,y_N)]. A sample polygon with 6 sides (N=6) is shown below.

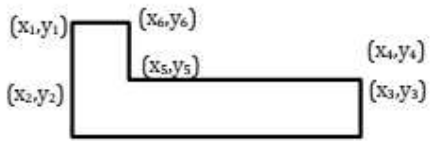

Figure 3 - Polygon

Write a python function that takes a list of N tuples as input and returns the perimeter of the polygon. Remember that your code should work for any value of N.

Hint: A perimeter is the sum of all sides of a polygon.

Solution:

```
def perimeter(listing):
    leng=len(listing)
    perimeter=0;
    for i in range(0,leng-1):
        dist = (((listing[i][0]-listing[i+1][0])**2)+
                ((listing[i][1]-listing[i+1][1])**2))**0.5
        perimeter = perimeter + dist
    perimeter = perimeter + (((listing[0][0]-listing[leng-1][0])**2)
                             +((listing[0][1]-listing[leng-1][1])**2))**0.5
    return perimeter

L = [(1,3), (2,7), (3,9), (-1,8)]
print(perimeter(L))
```

Activity 5:

Imagine two sets A and B containing numbers. Without using built-in set functionalities, write your own function that receives two such sets and returns another set C which is a symmetric difference of the two input sets. (A symmetric difference between A and B will return a set C which contains only those items that appear in one of A or B. Any items that appear in both sets are not included in C). Now compare the output of your function with the following built-in functions/operators.

- ✓ ***A.symmetric_difference(B)***
- ✓ ***B.symmetric_difference(A)***
- ✓ ***A ^ B***
- ✓ ***B ^ A***

Solution:

```
#Function defined
def symmDiff(a,b):
    e=set()   #empty set
    for i in a:   #for loop used to access in a
        if i  not in b:
            e.add(i)
    for i in b: #for loop used to access in b
        if i  not in a:
            e.add(i)
    return e


set1={0,1,2,4,5}
set2={4,5,7,8,9}
print(symmDiff(set1,set2))

#verification using inbuilt function
print(set1.symmetric_difference(set2))
print(set2.symmetric_difference(set1))
print(set1^set2)
print(set2^set1)
```

Activity 6:

Create a Python program that contains a dictionary of names and phone numbers. Use a tuple of separate first and last name values for the key field. Initialize the dictionary with at least three names and numbers. Ask the user to search for a phone number by entering a first and last name. Display the matching number if found, or a message if not found.

Solution:

```
sample={("sohaib","ali"):"0246585468445", ("aib","li"):"02465854645",
        ("sib","ai"):"0246585468445",}
firstName = input("enter first name")
lastName = input("enter last name")

searchTuple = (firstName, lastName)
if searchTuple in sample:
    print(sample[searchTuple])
else:
    print("name not found")
```

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficult and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab

Lab Task 1:

Create two lists based on the user values. Merge both the lists and display in sorted order.

Lab Task 2:

Repeat the above activity to find the smallest and largest element of the list. (Suppose all the elements are integer values)

Lab Task 3:

The derivate of a function f(x) is a measurement of how quickly the function f changes with respect to change in its domain x. This measurement can be approximated by the following relation,

$$\frac{d}{dx}f(x) = \frac{f(x+h) - \mathrm{f}(x)}{h}$$

Where h represents a small increment in x. You have to prove the following relation

$$\frac{d}{dx}\sin(x) = \cos(x)$$

Imagine x being a list that goes from –pi to pi with an increment of 0.001. You can approximate the derivative by using the following approximation,

$$\frac{d}{dx}\sin(x) = \frac{sin(x+h) - \sin(x)}{h}$$

*In your case, assume h = 0.001. That is at each point in x, compute the right hand side of above equation and compare whether the output value is equivalent to cos(x). Also print the corresponding values of ( ) and cos(x) for every point. Type ''from math import *'' at the start of your program to use predefined values of pi, and sin and cos functions. What happens if you increase the interval h from 0.001 to 0.01 and then to 0.1?*

Lab Task 4:

For this exercise, you will keep track of when our friend's birthdays are, and be able to find that information based on their name. Create a dictionary (in your file) of names and birthdays. When you run your program it should ask the user to enter a name, and return the birthday of that person back to them. The interaction should look something like this:

>>> Welcome to the birthday dictionary. We know the birthdays of:
Albert Einstein
Benjamin Franklin
Ada Lovelace

>>> Who's birthday do you want to look up?
Benjamin Franklin
>>> Benjamin Franklin's birthday is 01/17/1706.

Lab Task 5:

Create a dictionary by extracting the keys from a given dictionary

Write a Python program to create a new dictionary by extracting the mentioned keys from the below dictionary.

Given dictionary:

```
sample_dict = {
   "name": "Kelly",
   "age": 25,
   "salary": 8000,
   "city": "New york"}

# Keys to extract
keys = ["name", "salary"]
```

Expected output:

{'name': 'Kelly', 'salary': 8000}