USMAN INSTITUTE OF TECHNOLOGY

Department of Computer Science
CS321 Artificial Intelligence

Lab# 01
Python Programming a Review

Objective:

This lab provides a comprehensive revision of basic programming concepts and their implementation in Python. The lab also revises some of the implementations specific to Python programming

Name of Student: _____

Roll No: _____ **Sec.** _____

Date of Experiment: _____

Marks Obtained/Remarks: _____

Signature: _____

Lab 01: Introduction to Python Programming

Introduction to Python

Python is a high-level, interpreted, interactive and object-oriented scripting language. Python is designed to be highly readable. It uses English keywords frequently where as other languages use punctuation, and it has fewer syntactical constructions than other languages.

Python is Interpreted: Python is processed at runtime by the interpreter. You do not need to compile your program before executing it.

Python is Interactive: You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.

Python is Object-Oriented: Python supports Object-Oriented style or technique of programming that encapsulates code within objects.

Python is a Beginner's Language: Python is a great language for the beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.

Python Program

If you are running new version of Python, then you would need to use print statement with parenthesis as in **print ("Hello, Python!")**. However, in Python version 2.4.x, you do not need the parenthesis. The above line produces the following result:

```
Hello, Python!
```

Quotation in Python

Python accepts single ('), double (") and triple (''' or """) quotes to denote string literals, as long as the same type of quote starts and ends the string. The triple quotes are used to span the string across multiple lines. For example, all the following are legal

```
word = 'word'
sentence = "This is a sentence."
paragraph = """This is a paragraph. It is
made up of multiple lines and sentences."""
```

Comments in Python

A hash sign (#), that is not inside a string literal, begins a comment. All characters after the # and up to the end of the physical line are part of the comment and the Python interpreter ignores them.

Assigning Values to Variables

Python variables do not need explicit declaration to reserve memory space. The declaration happens automatically when you assign a value to a variable. The equal sign (=) is used to assign values to variables.

The operand to the left of the = operator is the name of the variable and the operand to the right of the = operator is the value stored in the variable.

```
counter = 100          # An integer assignment
miles   = 1000.0       # A floating point
name    = "Usman"      # A string
print (counter)
print (miles); print (name)
```

Data Types

The data stored in memory can be of many types. For example, a person's age is stored as a numeric value and his or her address is stored as alphanumeric characters. Python has various standard data types that are used to define the operations possible on them and the storage method for each of them. Python has five standard data types

- Numbers
- String
- List
- Tuple
- Dictionary

Python Numbers

Number data types store numeric values. Number objects are created when you assign a value to them.

```
var1 = 1
var2 = 10
```

Python Strings

Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Python allows for either pairs of single or double quotes. Subsets of strings can be taken using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the string and working their way from -1 at the end.

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

This will produce the following result

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

Python Lists

The list is the most versatile data type available in Python, which can be written as a list of comma-separated values (items) between square brackets. Important thing about a list is that the 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 ]
print ("list1[0]: ", list1[0])
print ("list2[1:5]: ", list2[1:5])
```

When the above code is executed, it produces the following result

```
list1[0]:  physics
list2[1:5]:  [2, 3, 4, 5]
```

Updating Lists

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

```
list = ['physics', 'chemistry', 1997, 2000]
print ("Value available at index 2 : ", list[2])

list[2] = 2001
print ("New value available at index 2 : ", list[2])
```

Delete List Elements

To remove a list element, you can use either the del statement if you know exactly which element(s) you are deleting. You can use the remove() method if you do not know exactly what is the index of the item to delete.

```
list = ['physics', 'chemistry', 1997, 2000]

print (list)
del list[2]
print ("After deleting value at index 2 : ", list)
```

Python Tuples

A tuple is another sequence data type that is similar to the list. A tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parenthesis. The main difference between lists and tuples are − Lists are enclosed in brackets ([ ]) and their elements and size can be changed, while tuples are enclosed in parentheses (( )) and cannot be updated. Tuples can be thought of as read-only lists.

```
tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  )
tinytuple = (123, 'john')
print (tuple)           # Prints complete tuple
print (tuple[0])        # Prints first element of the tuple
print (tuple[1:3])      # Prints elements starting from 2nd till 3rd
print (tuple[2:])       # Prints elements starting from 3rd element
print (tinytuple * 2)   # Prints tuple two times
print (tuple + tinytuple) # Prints concatenated tuple
```

This produces the following result

```
('abcd', 786, 2.23, 'john', 70.200000000000003)
abcd
(786, 2.23)
```

```
(2.23, 'john', 70.200000000000003)
(123, 'john', 123, 'john')
('abcd', 786, 2.23, 'john', 70.200000000000003, 123, 'john')
```

The following code is invalid with tuple, because we attempted to update a tuple, which is not allowed. Similar case is possible with lists

```
tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  )
list = [ 'abcd', 786 , 2.23, 'john', 70.2  ]
tuple[2] = 1000    # Invalid syntax with tuple
list[2] = 1000     # Valid syntax with list
```

Python Dictionary

Python's dictionaries are kind of hash-table type. They consist of key-value pairs. A dictionary key can be almost any Python type, but are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object. Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([]).

```
dict = {}
dict['one'] = "This is one"
dict[2]     = "This is two"
tinydict = {'name': 'john','code':6734, 'dept': 'sales'}

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

This produces the following result

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

Python 3 - Decision Making

Decision-making is the anticipation of conditions occurring during the execution of a program and specified actions taken according to the conditions. Decision structures evaluate multiple expressions, which produce TRUE or FALSE as the outc ome. You need to determine which action to take and which statements to execute if the outcome is TRUE or FALSE otherwise. Following is the general form of a typical decision making structure found in most of the programming languages.

IF Statement

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

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

```
var1 = 100
```

```
if var1:
   print ("1 - Got a true expression value")
   print (var1)
var2 = 0
if var2:
   print ("2 - Got a true expression value")
   print (var2)
print ("Good bye!")
```

IF...ELIF...ELSE Statements

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

```
if expression:
   statement(s)

else:
   statement(s)
```

```
amount = int(input("Enter amount: "))

if amount<1000:
   discount = amount*0.05
   print ("Discount",discount)
else:
   discount = amount*0.10
   print ("Discount",discount)

print ("Net payable:",amount-discount)
```

The elif statement allows you to check multiple expressions for TRUE and execute a block of code as soon as one of the conditions evaluates to TRUE.
Similar to the else, the elif statement is optional. However, unlike else, for which there can be at the most one statement, there can be an arbitrary number of elif statements following an if.

```
if expression1:
   statement(s)
elif expression2:
   statement(s)
elif expression3:
   statement(s)
else:
   statement(s)
amount = int(input("Enter amount: "))

if amount<1000:
   discount = amount*0.05
   print ("Discount",discount)
elif amount<5000:
   discount = amount*0.10
   print ("Discount",discount)
```

```
else:
   discount = amount*0.15
   print ("Discount",discount)

print ("Net payable:",amount-discount)
```

Loops

In general, statements are executed sequentially − The first statement in a function is executed first, followed by the second, and so on. 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 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.

While Loop Statements

A while loop statement in Python programming language repeatedly executes a target statement as long as a given condition is true.

```
count = 0
while (count < 9):
   print ('The count is:', count)
   count = count + 1

print ("Good bye!")
```

For Loop Statements

The for statement in Python has the ability to iterate over the items of any sequence, such as a list or a string.

```
>>> for var in list(range(5)):
   print (var)
```

```
for letter in 'Python':     # traversal of a string sequence
   print ('Current Letter :', letter)
print()
fruits = ['banana', 'apple',  'mango']
for fruit in fruits:        # traversal of List sequence
   print ('Current fruit :', fruit)

print ("Good bye!")
```

```
Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : h
Current Letter : o
Current Letter : n

Current fruit : banana
Current fruit : apple
Current fruit : mango
```

```
Good bye!
```

Functions

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

As you already know, Python gives you many built-in functions like print(), etc. but you can also create your own functions. These functions are called user-defined functions.

Defining a Function

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

- 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. You can also define parameters inside 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, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

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

Calling a Function

Defining a function 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. Following is an example to call the printme() function.

```
# Function definition is here
def printme(str):
   "This prints a passed string into this function"
   print (str)
   return
# Now you can call printme function
printme("This is first call to the user defined function!")
```

```
printme("Again second call to the same function")
```

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.

```
# Function definition is here
def changeme( mylist ):
   "This changes a passed list into this function"
   print ("Values inside the function before change: ", mylist)
   mylist[2]=50
   print ("Values inside the function after change: ", mylist)
   return

# Now you can call changeme function
mylist = [10,20,30]
changeme( mylist )
print ("Values outside the function: ", mylist)
```

```
Values inside the function before change:  [10, 20, 30]
Values inside the function after change:  [10, 20, 50]
Values outside the function:  [10, 20, 50]
```

```
# Function definition is here
def changeme( mylist ):
   "This changes a passed list into this function"
   mylist = [1,2,3,4] # This would assign new reference in mylist
   print ("Values inside the function: ", mylist)
   return

# Now you can call changeme function
mylist = [10,20,30]
changeme( mylist )
print ("Values outside the function: ", mylist)
```

```
Values inside the function:  [1, 2, 3, 4]
```

```
Values outside the function:  [10, 20, 30]
```

Function Arguments

You can call a function by using the following types of formal arguments.

- Required arguments
- Keyword arguments
- Default arguments
- Variable-length arguments

The Return Statement

The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return none. All the examples given below are not returning any value. You can return a value from a function.

```
# Function definition is here
def sum( arg1, arg2 ):
   # Add both the parameters and return them."
   total = arg1 + arg2
   print ("Inside the function : ", total)
   return total
# Now you can call sum function
total = sum( 10, 20 )
print ("Outside the function : ", total )
```

```
Inside the function :  30
Outside the function :  30
```

Object Oriented Programming

1. The __init__( ) Method

The __init__( ) method is profound for two reasons. Initialization is the first big step in an object's life; every object must be initialized properly to work properly. The second reason is that the argument values for __init__( ) can take on many forms. Because there are so many ways to provide argument values to __init__( ), there is a vast array of use cases for object creation. We take a look at several of them. We want to maximize clarity, so we need to define an initialization that properly characterizes the problem domain. Before we can get to the __init__( ) method, however, we need to take a look at the implicit class hierarchy in Python, glancing, briefly, at the class named object. This

will set the stage for comparing default behavior with the different kinds of behavior we want from our own classes.
In this example, we take a look at different forms of initialization for simple objects (for example, playing cards). After this, we can take a look at more complex objects, such as hands that involve collections and players that involve strategies and states.
Python is a multi-paradigm programming language. Meaning, it supports different programming approach.
One of the popular approach to solve a programming problem is by creating objects. This is known as Object-Oriented Programming (OOP).

An object has two characteristics:

- attributes
- behavior

Let's take an example:

Parrot is an object,

- name, age, color are attributes
- singing, dancing are behavior

The concept of OOP in Python focuses on creating reusable code. This concept is also known as DRY (Don't Repeat Yourself). In Python, the concept of OOP follows some basic principles:

1	***Inheritance***	A process of using details from a new class without modifying existing class.
2	***Encapsulation***	Hiding the private details of a class from other objects.
3	***Polymorphism***	A concept of using common operation in different ways for different data input.

Python Object Inheritance
Inheritance is the process by which one class takes on the attributes and methods of another. Newly formed classes are called child classes, and the classes that child classes are derived from are called parent classes. It's important to note that child classes override or extend the functionality (e.g., attributes and behaviors) of parent classes. In other words, child classes inherit all of the parent's attributes and behaviors but can also specify different behavior to follow. The most basic type of class is an object, which generally all other classes inherit as their parent. When you define a new class, Python 3 it implicitly uses object as the parent class. So the following two definitions are equivalent:

Exercise 1: Write a class of Dog, each dog must be of species type mammal. Each dog has its name and age. The class can have method for description () and sound () which dog produces. Create an object and perform some operations.

```
class Dog:

    # Class Attribute
    species = 'mammal'

    # Initializer / Instance Attributes
    def __init__(self, name, age):
```

```
        self.name = name
        self.age = age

    # instance method
    def description(self):
        return "{} is {} years old".format(self.name, self.age)

    # instance method
    def speak(self, sound):
        return "{} says {}".format(self.name, sound)

# Instantiate the Dog object
razer = Dog("Razer", 6)

# call our instance methods
print(razer.description())
print(razer.speak("Woof Woof"))
```

Exercise 2: Write a class of Dog, each dog must be of species type mammal. Each dog has its name and age. The class can have method for description () and sound () which dog produces. Now this time you need to create two sub classes of Dogs one is Bull Dog and other is Russell Terrier Create few objects and perform some operations including the inheritance.

```
# Parent class
class Dog:

    # Class attribute
    species = 'mammal'

    # Initializer / Instance attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # instance method
    def description(self):
        return "{} is {} years old".format(self.name, self.age)

    # instance method
    def speak(self, sound):
        return "{} says {}".format(self.name, sound)


# Child class (inherits from Dog class)
class RussellTerrier(Dog):
    def run(self, speed):
        return "{} runs {}".format(self.name, speed)

# Child class (inherits from Dog class)
class Bulldog(Dog):
    def run(self, speed):
        return "{} runs {}".format(self.name, speed)
```

```
# Child classes inherit attributes and
# behaviors from the parent class
thunder = Bulldog("Thunder", 9)
print(thunder.description())

# Child classes have specific attributes
# and behaviors as well
print(thunder.run("slowly"))

spinter = Bulldog("Spinter", 12)
print(spinter.description())
print(spinter.run("fast"))

roger = RussellTerrier("Roger", 5)
```

Exercise 3: Extending question number 2, now we need to check that either the different dog classes and their objects link with each other or not. In this case we need to create a method to find either it's an instance of each other objects or not.

```
# Parent class
class Dog:

    # Class attribute
    species = 'mammal'

    # Initializer / Instance attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # instance method
    def description(self):
        return "{} is {} years old".format(self.name, self.age)

    # instance method
    def speak(self, sound):
        return "{} says {}".format(self.name, sound)


# Child class (inherits from Dog() class)
class RussellTerrier(Dog):
    def run(self, speed):
        return "{} runs {}".format(self.name, speed)


# Child class (inherits from Dog() class)
class Bulldog(Dog):
    def run(self, speed):
        return "{} runs {}".format(self.name, speed)


# Child classes inherit attributes and
# behaviors from the parent class
thunder = Bulldog("Thunder", 9)
print(thunder.description())
```

```
# Child classes have specific attributes
# and behaviors as well
print(thunder.run("slowly"))

# Is thunder an instance of Dog()?
print(isinstance(thunder, Dog))

# Is thunder_kid an instance of Dog()?

thunder_kid = Dog("ThunderKid", 2)
print(isinstance(thunder, Dog))

# Is Kate an instance of Bulldog()
Kate = RussellTerrier("Kate", 4)
print(isinstance(Kate, Dog))

# Is thunder_kid and instance of kate?
print(isinstance(thunder_kid, Kate))
print("Thanks for understanding the concept of OOPs")
```

NOTE:
Make sense? Both thunder_kid and Kate are instances of the Dog() class, while Spinter is not an instance of the Bulldog() class. Then as a sanity check, we tested if kate is an instance of thunder_kid, which is impossible since thunder_kid is an instance of a class rather than a class itself—hence the reason for the TypeError.

Student Exercise

Lab Task: Go through and attempt all the notebooks provided to you along with this lab.

Homework: Attempt all the exercises provided to you inside the homework folder. Your understanding of Python will be evaluated through a Quiz before the next lab.