Lab 01
Python concepts

Objectives:
This lab is an introductory session on Python. The lab will equip students with necessary concepts needed to explore and analyze cryptographic 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 practice activity, students may explore following sites to gain an insight about python programming and its fundamentals.

- https://www.w3schools.com/python/default.asp
- https://www.geeksforgeeks.org/how-to-learn-python-from-scratch/

1) Practice based Lab Activities

Python Basics

Prior to getting into the variables, spacing, strings, and loops, some useful information that need to be known. I will cover a few of the basics you will need to know. Names in Python are case sensitive and cannot start with a number. They can contain letters, numbers, and underscores. Some examples include:

- alice
- Alice
- _alice
- _2_alice_
- alice_2

The language includes a number of reserved words, such as and, assert, break, class, continue, def, del, elif, else, except, exec, finally, for, from, global, if, import, in, is, lambda, not, or, pass, print, raise, return, try, while.

One major difference between Python and many other languages is that whitespace is meaningful, and many of your early errors probably will be a result of indentation being misused. A newline in Python ends the line of code unless you use the backslash (\) character. You will find that you need to use consistent indentation throughout your code. The first line of code with less indentation is outside a code block, whereas the lines with more indentation start a nested block.

As with other languages, Python allows you to comment your code. Commenting your code should be considered a must. Some of the comments in the code used in this book have been removed to shorten the code examples, but you will be well-served by comments as you revisit previously written code. In Python, comments start with #; everything following the pound sign will be ignored.

Using Variables

The programs you write throughout will be need to store values so they can be used at a later time. Values are stored in variables; the use of a variable is signified by the = sign, which is also known as the assignment operator.

For example, if you wanted to store the value 21 in a variable named age, enter age = 21 into the shell:

```
>>> age = 21
```

Variables can be overwritten or used in calculations. Explore the following by typing it into your shell:

```
>>> age + 9
30
>>> age
21
>>> age = age + 9
>>> age
```

```
30
```

Python doesn't enforce any variable naming conventions, but you should use names that reflect the type of data that is being stored. It is important to know that in Python, variable names are case sensitive, so age and Age represent two different variables. This overview shows the use of variables with numbers; we will examine the use of variables and strings next.

Using Strings

Strings allow you to use text in programming languages. In cryptography, you use strings a great deal to convert plaintext to ciphertext and back. These values are stored in variables much like the numeric examples used in the preceding section. Strings in Python are stored using the single quote (‘) or double quote (“); it does not matter which quote you use as long as they are matching. To see an example, type the following into the shell:

```
>>> name = 'John'

>>> name

'John'
```

The single quotes are not part of the string value. Python only evaluates the data between the quotes. If you wanted to set a variable to an empty string, you would use the following:

```
>>> name = ''
```

In Python, you can concatenate multiple strings together by using the plus (+) operator. See the following example:

```
>>> first = 'John'

>>> last = 'Doe'

>>> name = first + last

>>> name

'JohnDoe'
```

Notice that the strings concatenate exactly the way they store values; if you need a word separation, plan for the use of spaces.

Introducing Operators

You have now seen the + operator used in both numeric and string variables. Other operators can be broken down into the following categories: arithmetic, comparison, logical, assignment, bitwise, membership, and identity. Understanding Arithmetic Operators Arithmetic operators perform mathematical calculations (see Table 1.1).

Table 1.1: Arithmetic Operators

OPERATOR	DESCRIPTION	EXAMPLE
+	Addition	10 + 20 will give 30
-	Subtraction	10 - 5 will give 5
*	Multiplication	10 * 10 will give 100
/	Division	10 / 2 will give 5
%	Modulus	20 % 10 will give 0
**	Exponent	10**2 will give 100
//	Floor Division	9//2 is equal to 4 and 9.0//2.0 is equal to 4.0

Most of these operators work precisely the way you expect from other programming languages. It is critical to our exploration to examine modular (%) arithmetic in detail as it plays an essential role in cryptography. We write things like $28 \equiv 2 \pmod{26}$, which is read out loud as "28 is equivalent to 2 mod 26." While not entirely accurate, modular arithmetic focuses on the remainder. In our example, 26 divides into 28 one time with two remaining, or x(mod p) as the remainder when x divides into p.

We can say that $a \equiv b \pmod{q}$ when $a-b$ is a multiple of q. So $123 \equiv 13 \pmod{11}$ because $123-13=110=11 \cdot 10$. Now I've heard folks describe the remainder of x mod p, which does refer to the unique number between 0 and $p-1$, which is equivalent to x. An infinite sequence of numbers are equivalent to 53(mod13), but out of that continuous sequence, there is precisely one number that is positive and smaller than 13. In this case, it's 1. In a math setting, I would call 1 the canonical representative of the equivalence class containing 53 modulo 13. Table 1.2 shows some examples to help you wrap your head around the concepts; they will be presented again when you examine ciphers and explore finding the modular inverse in cryptographic math

Table 1.2: Arithmetic Operator Examples

EXPRESSION	DESCRIPTION	SYNTAX
28 (mod 26)	28 is equivalent to 2 mod 26	28 % 26
29 (mod 26)	29 is equivalent to 3 mod 26	29 % 26
30 (mod 26)	30 is equivalent to 4 mod 26	30 % 26

One additional feature of Python is the use of the multiplication operator with strings. The feature can be advantageous when you need to create specifically formatted strings that may be used in some attacks such as buffer overflows. One example can be examined using the print function, as shown here:>>> # Python 2.7

```
>>> print 'a' * 25
aaaaaaaaaaaaaaaaaaaaaaaaa
>>> # Python 3.5

>>> print ('a' * 25)

aaaaaaaaaaaaaaaaaaaaaaaaa
```

Understanding Comparison Operators

The comparison operators, also known as relational operators, compare the oper ands (values) on either side and return true or false based on the condition (see Table 1.3)

Table 1.3: Comparison Operators

OPERATOR	DESCRIPTION	EXAMPLE
==	Compares two operands to see if they are equal; if the values are equal, it returns true.	(10 == 20) is not true
!=	Compares two operands to see if they are not equal; if the values are not equal, it returns true.	(10 != 20) is true
<>	Compares two operands to see if they are not equal; if the values are not equal, it returns true.	(10 <> 20) is true
>	If the operand on the left is greater than the operand on the right, the operation returns true.	(10 > 5) is true
<	If the operand on the right is greater than the operand on the left, the operation returns true.	(10 < 20) is true
>=	If the operand on the right is equal to or less than the value on the left, the condition returns true.	(10 >= 5) is true
<=	If the operand on the left is equal to or less than the value on the right, the condition returns true.	(5 <= 10) is true

Understanding Logical Operators The logical operators include and, or, and not (see Table 1.4).

Table 1.4: Logical Operators

OPERATOR	DESCRIPTION	EXAMPLE
and (logical AND)	If both the operands evaluate to 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.

Understanding Assignment Operators You first explored the assignment operators when we covered variables. In addition to using the equal sign, Python offers many assignments that work as a shorthand for more extended tasks. To get an understanding of how this works, examine Table 1.5.

Table 1.5: Assignment Operators

OPERATOR	DESCRIPTION	EXAMPLE
=	Assigns values from right-side operands to left-side operands.	`c = a + b` assigns value of `a + b` into `c`
+= (add AND)	Adds the right operand to the left operand and assigns the result to the left operand.	`c += a` is equivalent to `c = c + a`
-= (subtract AND)	Subtracts the right operand from the left operand and assigns the result to the left operand.	`c -= a` is equivalent to `c = c - a`
*= (multiply AND)	Multiplies the right operand with the left operand and assigns the result to the left operand.	`c *= a` is equivalent to `c = c * a`
/= (divide AND)	Divides the left operand with the right operand and assigns the result to the left operand.	`c /= a` is equivalent to `c = c /`
%= (modulus AND)	Takes modulus using two operands and assigns the result to the left operand.	`c %= a` is equivalent to `c = c % a`
**= (exponent AND)	Performs exponential (power) calculation on operators and assigns the value to the left operand.	`c **= a` is equivalent to `c = c ** a`
//= (floor division)	Performs floor division on operators and assigns the value to the left operand.	`c //= a` is equivalent to `c = c // a`

Understanding Bitwise Operators The bitwise operators work on bits and perform bit-by-bit operations (see Table 1.6). Assuming a = 60 and b = 13, the binary format of a and b will be as follows: a = 0011 1100 b = 0000 1101

Table 1.6: Bitwise Operators

OPERATOR	DESCRIPTION	EXAMPLE
& (binary AND)	Copies a bit to the result if it exists in both operands.	(a & b) (means 0000 1100)
| (binary OR)	Copies a bit if it exists in either operand.	(a | b) = 61 (means 0011 1101)
^ (binary XOR)	Copies the bit if it is set in one operand but not both.	(a ^ b) = 49 (means 0011 0001)
~ (binary One Complement)	This operator is unary and has the effect of "flipping" bits.	(~a) =-61 (means 1100 0011 in two's complement form due to a signed binary number.
<< (binary Left Shift)	The left operand's 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 operand's value is moved right by the number of bits specified by the right operand.	a >> 2 = 15 (means 0000 1111)

Understanding Membership Operators

Membership operators test for membership in a sequence, such as strings, lists, or tuples. There are two membership operators, in and not in, as shown in Table 1.7

Table 1.7: 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 find 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.

Understanding Identity Operators

Table 1.8: 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).

Using Conditionals

You will need to evaluate various conditions as you are writing your code; conditional statements in Python perform different actions or computations and decide whether a condition evaluates to true or false. We use the IF statement for our Boolean test. The syntax is as follows:

```
If expression
   Statement
Else
   Statement
```

To use it in an actual program, type the following:

```
 # Python 3.5
 for i in range(1,5):
     if i == 2:
         print ('I found two')
     print (i)
```

In addition to the IF statement, Python also makes use of ELSE and ELIF. ELSE will capture the execution if the condition is false. ELIF stands for else if; this gives us a way to chain several conditions together. Examine the following:

```
>>> # Python 3.5
>>> for i in range(1,5):
>>>     if i == 1:
>>>         print ('I found one')
>>>     elif i == 2:
>>>         print ('I found two')
>>>     elif i == 3:
>>>       Print ('I found three')
>>>     else
>>>       Print ('I found a number higher than three')
>>>     print (i)
```

Using Loops

When you use scripting or programming languages, you can perform a set of statements in multiple repetitions. Loops give us the ability to run logic until a specific condition is met. This section shows the for loop, the while loop, the continue statement, the break statement, and the else statement. Unlike other programming languages that signify the end of the loop by using keywords or brackets, Python uses line indentations. You need to focus on how your syntax is spaced when writing and testing your code.

for

The for loop is used to iterate over a set of statements that need to be repeated n number of times. The for statement can be used to execute as a counter in a range of numbers. The following syntax prints out the values 1, 2, 3, 4, 5, 6, 7, 8, 9:

```
>>> # Python 3.5
>>> for i in range(1, 10):
>>>     print (i)
```

The for loop can also execute against the number of elements in an array. Examine the following snippet, which produces the result 76:

```
                    >>> # Python 3.5
>>> numbers = [1, 5, 10, 15, 20, 25]
```

```
>>> total = 0
>>> for number in numbers:
>>>     total = total + number
>>> print (total)
```

As with numbers, the same technique is useful against string arrays. The following outputs three names—Eden, Hayden, and Kenna:

```
>>> # Python 3.5
>>> all_kids = ["Eden&", "Hayden&", "Kenna&"]
>>> for kid in all_kids:
>>>     print kid
Eden
Hayden
Kenna
```

while

The while loop is used to execute a block of statements while a condition is true. As with the for loop, the block of statements may be one or more lines. The indentation of the following lines defines the block. Once a condition becomes false, the execution exits the loop and continues. Before printing the final statement, the following snippet prints 0, 1, 2, 3, 4:

```
>>> # Python 3.5
>>> count = 0 >>> while (count < 5):
>>> print count
>>> count = count + 1
>>> print ("The loop has finished.")
```

continue

The continue statement is used to tell Python to skip the remaining statements in the current loop block and continue to the next iteration. The following snippet will produce an output of 1, 3, 4. The continue statement skips printing when i equals 2:

```
>>> # Python 3.5
>>> for i in range(1,5):
>>>     if i == 2:
>>>         continue
>>>     print (i)
```

break

The break statement exits out from a loop. The following snippet produces an output of 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11. Once i = 12, the loop is abandoned:

```
>>> # Python 3.5
>>> for i in range(1,15):
>>>     if i == 12:
>>>         break
>>>     print (i)
```

else

You can use the else statement in conjunction with the for and while loops to catch conditions that fail either of the loops. Notice that since the count variable equals 10 in the following example, the while loop does not execute. You should only see the two final print messages:

```
>>> # Python 3.5
 >>> count = 10
>>> while (count < 5):
 >>>     print (count)
 >>>     count = count + 1
 >>> else:
 >>>     print ("The count is greater than 5")
 >>> print ("The loop has finished.")
```

Using Files

we will be using external files to both read and write. We will be using the file function, which takes the file location and then performs an operation. Table 1.9 shows the operations available.

Table 1.9: File Operations

OPERATOR	DESCRIPTION
R	Open text file for reading. The stream is positioned at the beginning of the file.
r+	Open for reading and writing. The stream is positioned at the beginning of the file.
W	Truncate file to zero length or create text file for writing. The stream is positioned at the beginning of the file.
w+	Open for reading and writing. The file is created if it does not exist, otherwise it is truncated. The stream is positioned at the beginning of the file.
Append	Open for writing/appending to the file. The file is created if it does not exist. The stream is positioned at the end of the file. Subsequent writes to the file will always end up at the current end of file.
a+	Open for reading and writing. The file is created if it does not exist. The stream is positioned at the end of the file. Subsequent writes to the file will always end up at the current end of file.

Perform following by creating .txt file and show it content after execution.

```
>>> # Python 3.5
 >>> f = open('dictionary.txt', 'r')
 >>> words = [word.strip() for word in f]
 >>> f.close()
```

Understanding Python Semantics

Assume you have assigned y an object. Examine the following:

x = y does not make a copy of the object y references.
x = y creates a reference to the object that y references.

This may be confusing but it is an important concept, so let's look at an example:

```
>>> # Python 3.5
>>> a = [1, 2, 3]     # a is a reference to the list [1, 2, 3]
>>> b = a        # b now references a
>>> a.append(4)    # appends the number 4 to the list a references
>>> print (b)        # print what b references
```

```
[1, 2, 3, 4]
```

Why does this happen? When you type a command such as x = 3, an integer is created and stored in memory. A variable named x is created and references the memory location storing the value, which in this case is 3. When you say that the value of x is 3, what you are really saying is that x now refers to the integer 3 at a specific memory location. The data type that is created when you
assign the reference to x is of type integer. In Python, the data types include integer, float, and string (and tuple) and are immutable. You can change the value that is referenced by the memory
location. Here is an example:

```
>>> # Python 3.5
>>> x = 4
>>> x = x + 1
>>> print x
```

```
5
```

In the preceding example, when you increment x, you are actually looking up the reference of the name x and the value that it references is retrieved. When you add 1 to the value, you are producing a new data element 5, which is assigned to a fresh memory location with a new reference. The name x is then changed to point to the new reference. The old data, which contained 4, is then garbage collected if no name refers to it.

Sequence Types

Python has a number of sequence types. These include str, Unicode, list, tuple, buffer, and xrange. You were previously introduced to string literals, so let's examine the other ones. Unicode strings are similar to strings but are indicated in the syntax using a preceding "u" character: u'abc', u"def". Lists are assembled with square brackets, separating list items with commas: [1, 2, 3]. Tuples are constructed with the comma operator (not within square brackets), with or without enclosing parentheses, but an empty tuple must have the enclosing parentheses, such as a, b, c or (). A single-item tuple must have a trailing comma, such as (d,). A tuple is a simple immutable ordered sequence of items that can be of mixed types. An immutable object is an object that often represents a single logical structure of data. *Lists in Python are a mutable ordered sequence of list items that can be of mixed types as well. Mutable objects are those that allow their state (i.e., data that the variable holds) to change.* In Python, only dictionaries and lists are mutable objects.

Buffer objects are not directly supported by Python syntax, but can be created by calling the built-in function buffer(). They don't support concatenation or repetition.
Xrange objects are similar to buffers in that there is no specific syntax to cre ate them, but they are created using the xrange() function. They don't support slicing, concatenation, or repetition, and using in, not in, min(), or max() on them is inefficient.
Most sequence types support the following operations. The in and not in operations have the same priorities as the comparison operations. The + and * operations have the same priority as the corresponding numeric operations:

Tuples are defined using parentheses (and commas):

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
```

■ Lists are defined using square brackets (and commas):

```
>>> li = ["abc", 34, 4.34, 23]
```

■ Strings are defined using quotes (", ', or """):

```
>>> st = "Hello World"
>>> st = 'Hello World'
>>> st = """This is a multi-line string that uses triple quotes."""
```

You can access individual members of a tuple, list, or string using square bracket "array" notation. All of the objects are zero-based, which means that the first element in the group is in the zero position:

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
>>> tu[1] # Second item in the tuple.
'abc'
>>> li = ["abc", 34, 4.34, 23]
>>> li[1] # Second item in the list.
34
>>> st = "Hello World"
>>> st[1] # Second character in string.
'e'
```

22

One of the most interesting operations in Python that we will make use of is how the indices work. You can use both positive and negative indices to work with the element you need:

```
>>> t = (23, 'abc', 3.14, (2,3), 'def')
```

With a positive index, you count from the left, starting with 0:

```
>>> t[1]
'abc'
```

With a negative index, you count from the right, starting with –1. Examine the t assignment shown previously. You will notice that 3.14 is in the 3rd position on the right. Using the negative number, you are able to start with the right side and work your way left. Examine how the -3 works next:

```
>>> t[-3]
3.14
```

You can use a similar technique to parse a range of data:

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Return a copy of the container with a subset of the original members. Start copying at the first index, and stop copying before the second index:

```
>>> t[1:4]
('abc', 4.56, (2,3))
```

You can also use negative indices when slicing:

```
>>> t[1:-1]
('abc', 4.56, (2,3))
```

You can omit the first index to make a copy starting from the beginning of the container:

```
>>> t[:2]
(23, 'abc')
```

You can omit the second index to make a copy starting at the first index and going to the end of the container:

```
>>> t[2:]
(4.56, (2,3), 'def')
```

To make a copy of an entire sequence, you can use [:]:

```
>>> t[:]
(23, 'abc', 4.56, (2,3), 'def')
```

Note the difference between these two lines for mutable sequences:

```
>>> list2 = list1      # 2 names refer to 1 ref
                       # Changing one affects both
>>> list2 = list1[:]   # Two independent copies, two ref
```

To examine the content within lists, you use the in operator. Some examples include the following:

■ Boolean test whether a value is inside a container:

```
>>> t = [1, 2, 4, 5]
>>> 3 in t
False
>>> 4 in t
True
>>> 4 not in t
False
```

■ For strings, tests for substrings:

```
>>> a = 'abcde'
>>> 'c' in a
True
>>> 'cd' in a
True
>>> 'ac' in a
False
```

Be careful: the in keyword is also used in the syntax of for loops and list comprehensions. The + operator produces a new tuple, list, or string whose value is the concatenation of its arguments:

```
>>> (1, 2, 3) + (4, 5, 6)
(1, 2, 3, 4, 5, 6)
>>> [1, 2, 3] + [4, 5, 6]
[1, 2, 3, 4, 5, 6]
>>> "Hello" + " " + "World"
'Hello World'
```

The * operator produces a new tuple, list, or string that repeats the original content:

```
>>> (1, 2, 3) * 3
(1, 2, 3, 1, 2, 3, 1, 2, 3)
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> "Hello" * 3
'HelloHelloHello'
```

We will now take another look at tuples. As stated previously, tuples are immutable, which means you cannot change them. You can only make a fresh tuple and assign its reference to a previously used name:

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
>>> t[2] = 3.14
Traceback (most recent call last):
File "<pyshell#75>", line 1, in -topleveltu[ 2] = 3.14
TypeError: object doesn't support item assignment
```

```
>>> t = (23, 'abc', 3.14, (2,3), 'def')
```

Lists, on the other hand, are mutable; this means we can change the lists without having to reassign them. The variable will continue to point to the same memory reference when the assignment is complete. The downside is that lists are not as fast as tuples:

```
>>> li = ['abc', 23, 4.34, 23]
>>> li[1] = 45
>>> li
['abc', 45, 4.34, 23]
```

The following examples are operations that only apply to list objects:

```
>>> li = [1, 11, 3, 4, 5]
>>> li.append('a') # Our first exposure to method syntax
>>> li
[1, 11, 3, 4, 5, 'a']
>>> li.insert(2, 'i')
>>>li
[1, 11, 'i', 3, 4, 5, 'a']
```

Python has an extend() method that operates on lists in place; this operation is different than the plus (+) operator, which creates a fresh list with a new memory reference.

```
>>> li.extend([9, 8, 7])
>>>li
[1, 2, 'i', 3, 4, 5, 'a', 9, 8, 7]
```

You may find this a bit confusing as the extend() method takes a list as an argument, whereas the append() method takes a singleton as an argument:

```
>>> li.append([10, 11, 12])
>>> li
[1, 2, 'i', 3, 4, 5, 'a', 9, 8, 7, [10, 11, 12]]
```

Additionally, Python list objects can also use the index(), count(), remove(), reverse(), and sort() methods as shown here:

```
>>> li = ['a', 'b', 'c', 'b']
>>> li.index('b') # index of first occurrence
1
>>> li.count('b') # number of occurrences
2
>>> li.remove('b') # remove first occurrence
>>> li
['a', 'c', 'b']
>>> li = [5, 2, 6, 8]
>>> li.reverse() # reverse the list *in place*
>>> li
[8, 6, 2, 5]
>>> li.sort() # sort the list *in place*
>>> li
[2, 5, 6, 8]
>>> li.sort(some_function) # sort in place using user-defined comparison.
```

One thing to keep in mind when you are debating between using tuples versus lists is that the list objects are slower but are more powerful than tuples.
Lists can be modified and have many operations that can be used on them. You can convert between tuples and lists by using the list() and tuple() functions, as shown here:

```
li = list(tu)
 tu = tuple(li)
```

Dictionaries provide a way to store a mapping between a set of keys and a set of values. The keys can be any immutable type, whereas the values can be any type. A single dictionary can store a range of different types and allow you to define, modify, view, look up, and delete a key-value pair within the dictionary.

Here is an example of a dictionary:

```
 >>> d = {'user':'bozo', 'pswd':1234}
 >>> d['user']
'bozo'
 >>> d['pswd']
1234
 >>> d['bozo']
 Traceback (innermost last):
 File '<interactive input>' line 1, in ?
 KeyError: bozo
 >>> d = {'user':'bozo', 'pswd':1234}
 >>> d['user'] = 'clown'
 >>> d
{'user':'clown', 'pswd':1234}
 >>> d['id'] = 45
 >>> d
{'user':'clown', 'id':45, 'pswd':1234}
 >>> d = {'user':'bozo', 'p':1234, 'i':34}
 >>> del d['user'] # Remove one.
 >>> d
{'p':1234, 'i':34}
 >>> d.clear() # Remove all.
 >>> d
{}
 >>> d = {'user':'bozo', 'p':1234, 'i':34}
 >>> d.keys() # List of keys.
['user', 'p', 'i']
 >>> d.values() # List of values.
['bozo', 1234, 34]
 >>> d.items() # List of item tuples.
[('user','bozo'), ('p',1234), ('i',34)]
```

Introducing Custom Functions

Now that you understand strings, let's examine reusable code. In Python, you create a new function and assign it a name by using the def keyword. All functions in Python return results to the calling statement. Arguments are passed by assignment, and arguments and return types are not declared. When you pass arguments to a function, the values are assigned to locally scoped names. The assignment to argument names will not affect the caller since they are passed by assignment and not by reference. You will, however, affect the caller if you change a mutable argument.

Create a new file in your editor of choice and type the following:

```
 def myEnc(plaintext, key):
   return "ciphertext"
```

Save the file as MyFunctions.py. In the command line, type MyFunctions.py. You should notice that nothing happens. Now type the following:

```
 myEnc('hello','secret key')
ciphertext.
```

The script you created is an example of a function. The def keyword is used to define the function; as you might have guessed, myEnc is the name of the function. It takes two parameters or inputs and returns an output. We use functions to build logic we intend to use multiple times. The benefit of using functions is that if you decide you need to change the script, you can change it in one place and not have to search for other areas in which you performed the same logic. This will help in both troubleshooting and code maintenance. In Python, you can declare some arguments as optional. Examine the following segment of code, paying attention to the third and fourth arguments:

```
def func(a, b, c=10, d=100):
! print (a, b, c, d)
>>> func(1,2)
1 2 10 100
>>> func(1,2,3,4)
1,2,3,4
```

Something to watch out for is that all functions have a return value even if you do not provide a return line inside the code. Functions that do not provide a return line will return the special value of None. Note also that, unlike other languages, Python does not provide a way to overload a method, so you are not allowed to have two different functions with the same name and a different list of arguments. Some benefits to functions include that they can be used as arguments to functions, return values of functions, and be assigned to variables, and may contain parts of tuples, lists, and other objects.

Introducing Python Modules

Python modules are special packages that extend the language. You saw your first module when you used the import math call shown previously. You can examine the modules that are loaded by typing dir(). As shown with the math module, when a module is preinstalled, we can use the import command to upload it. To examine the modules that are preinstalled on your system, type the following:

```
>>> help()
help> modules
help> modules hashlib
q
```

The hashlib is a built-in module that is preinstalled that will allow you to run hash functions. we will explore the use of hashlib as a module. Type import hashlib, then in the terminal type hashlib. (enter the dot after hashlib), and press the Tab key. You should see a list of methods, as shown in Figure 1.4. You can explore these methods by typing the module and method followed by a question mark, such as hashlib.sha256?

Figure 1.4: Module methods

Creating a Reverse Cipher

For this next exercise, we build on what we have learned in this chapter by creating a function named reverseCipher that accepts one parameter: plaintext. We can then call our function and pass the plaintext into it and print out the return ciphertext. Our method modifies the plaintext so that the ciphertext is the complete

reverse:

```
def reverseCipher(plaintext):
   ciphertext = ''
   i = len(plaintext) - 1
   while i >= 0:
      ciphertext = ciphertext + plaintext[i]
      i = i - 1
   return ciphertext
plaintext = 'If you want to keep a secret, you must also hide it from
yourself.'
ciphertext = reverseCipher (plaintext)
print(ciphertext)
```

This code should produce the following result:

```
.flesruoy morf ti edih osla tsum uoy ,terces a peek ot tnaw uoy fI
```

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 happensautomatically 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 typesare 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)123123123123123123123123123123123123123123123124
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: The underlying type of a Python integer, irrespective of the base used to specify it, is called int:

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

```
>>> 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 decimalpoint. 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 tothe 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 zerothan 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 theopening 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'smemory 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, soPython assumes the next single quote, the one in parentheses which was intended to be part of thestring, 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 thestring 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 stringdifferently. 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 specialinterpretation 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 thenewlines 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 themto 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 theescape 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 interpretingliterally:

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 fromthe 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 stringare 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 beused 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 twovalues, True or False:

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

As you will see in upcoming content, 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 issometimes said to be "truthy," and one that is false in Boolean context is said to be "falsy." (You mayalso see "falsy" spelled "falsey.") The "truthiness" of an object of Boolean type is self-evident: Boolean objects that are equal to True aretruthy (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.

Solved Lab Activities

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 evaluatedin the same lab.

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	*10*	*Low*	*CLO-6*
5	*10*	*Low*	*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 thegiven 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 isprime 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 beforecoding 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 toa 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:

How you can import the hashlib library and then use the MD5 and SHA message digests on sample word string.

Solution:

Now that you understand the importance of Python modules, let's import the hashlib library and then use the MD5 and SHA message digests:

```
>>> import hashlib
>>> hashlib.md5('hello world'.encode()).hexdigest()
```

Output

```
'5eb63bbbe01eeed093cb22bb8f5acdc3'
```

```
>>> hashlib.sha512('hello world'.encode()).hexdigest()
```

Output

```
'309ecc489c12d6eb4cc40f50c902f2b4d0ed77ee511a7c7a9bcd3ca86d4cd86f
989dd35bc5ff499670da34255b45b0cfd830e81f605dcf7dc5542e93ae9cd76f'
```

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***

4) Home Lab Tasks

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 haveto 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 loopsfor 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 01
Python concepts

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, perform given tasks to gain an insight about python programming and its fundamentals.

Note:

Remaining activities will be considered as Home activities subjective to Lab session time limit

1) Some 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 toolin 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 toa 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 numbersList = [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 orduplicate 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 ListUsing 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 ListList =
[]
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 anitem 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 itremoves 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.

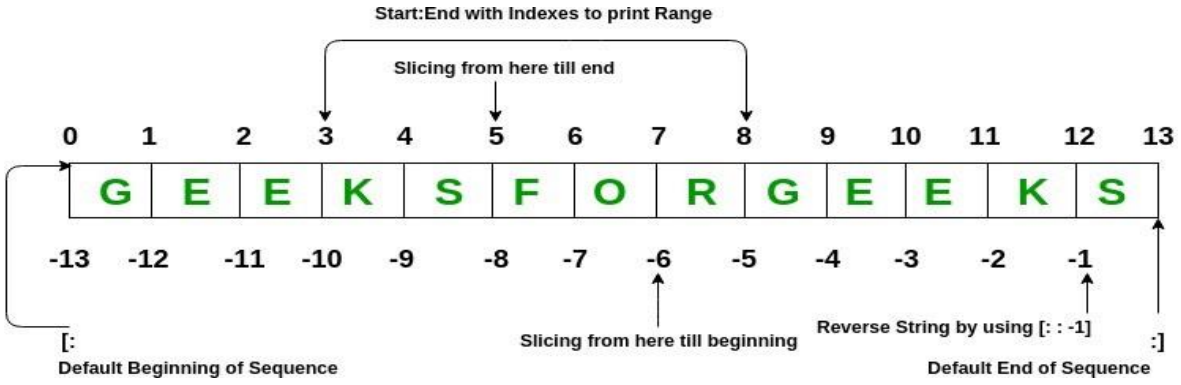

Figure 1 - List

```
# 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
```

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 andcan 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 byjust 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:

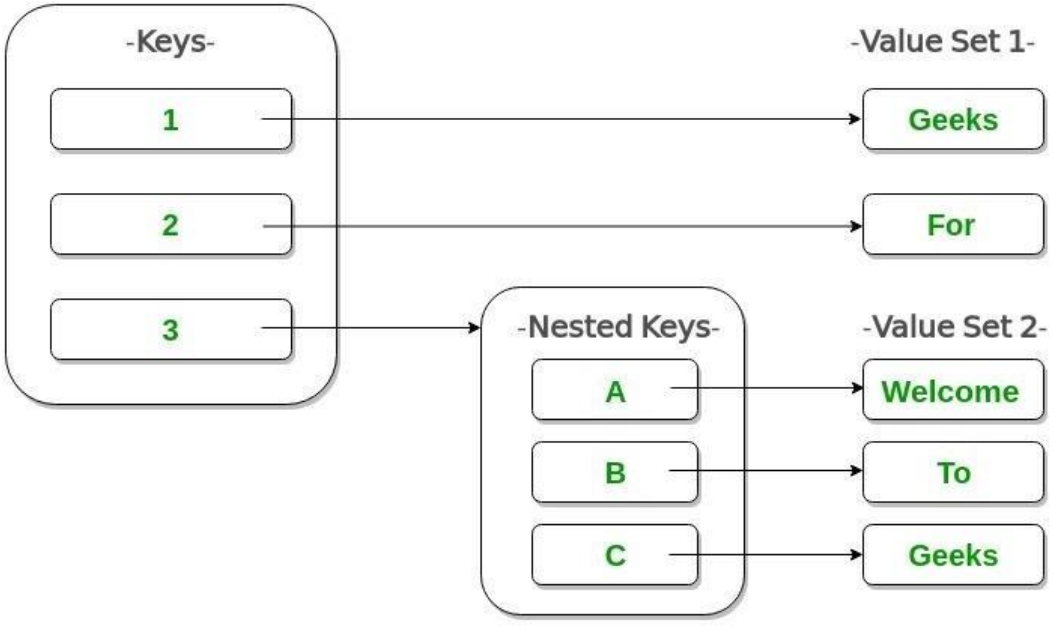

Figure 2 - Dictionary

```
# 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 timecan 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 squarebrackets.

```
# 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) Lab Practice Activities:

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:

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 5:

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")
```