PYTHON PROGRAMMING

UNIT - I

PYTHON P

INTRODUCTION TO PYTHON

- Python is an open source, object-oriented, high-level powerful programming language.
- Python is an easy to learn, powerful programming language.
- It is designed to be highly readable.
- Python is Interpreted i.e Processed at run time by the interpreter.

HISTORY OF PYTHON

- Python was developed by **Guido van Rossum** in the late eighties and early nineties at the National Research Institute for Mathematics and Computer Science in the Netherlands.
- The name Python was selected from "**Monty Python's Flying Circus**" which was a British comedy series created by the comedy group Monty Python and broadcast by the BBC from 1969 to 1974.
- Python is derived from many other languages, including ABC, Modula-3, C, C++,Algol-68, Small Talk, Unix shell, and other scripting languages.

FEATURES OF PYTHON

- **Open source:** Python is publicly available open source software.
- **Easy-to-Learn: Popular** (scripting/extension) language,clear and easy syntax, high-level data types and operations, design to read (more English like syntax) and write (shorter code compared to C, C++, and Java) fast.
- High-Level-Language.
- **Portable**-It means they are able to run across all major hardware and software platforms with few or no change in source code.
- **Object-Oriented:** It consists of features such as classes, inheritance, objects, and overloading.
- **Python is Interactive:** It has an interactive console where we get a Python prompt (command line) and interact with the interpreter directly to write.
- **Interpreted:** Python programs are interpreted, takes source code as input ,compiles (to portable byte-code) each statement and executes it immediately. No need to compiling or linking.
- **Extendable:** Python is often referred to as a "glue" language,i.e that it is capable to work in mixed-language environment. The interpreter is easily extended and can add a new built-in function written in C/C++/Java code.
- **Libraries:** Databases, web services, networking, numerical packages, graphical user interfaces, 3D graphics, others.

PYTHON PROGRAMMING

HOW PYTHON DIFFERS FROM OTHER LANGUAGE?

It is very easy to code.

C	C++	JAVA	PYTHON

#include <stdio.h>
int main()
{
printf("Hello world");
}
	#include <iostream>
	int main()
	{
	std::cout << "Hello world" << std::endl;
	return 0;
	}
		public class Hello
		{
		public static void main(String argv[])
		{
		
		System.out.println("Hello, World!");
		}
		}
			print ("Hello World")

VARIABLES

- A variable is a location in memory used to store some data (value).

PYTHON	C/JAVA
1. Declaration of variables is not required in Python.	1Declaration of variables is required.
2. Each variable can have values of different data type. Eg:a→can store any values of any data type	2. Each variable must have a unique data type. Eg:int a →It can store only integer data type.
3.The value of a variable and type may change during program execution. Eg:i=42 i='ajay'	3.Only the value can change. Eg:i=2 i=3

- Variables in Python follow the standard nomenclature of an alphanumeric name beginning in a letter or underscore.
- Variable names are case sensitive.
- Variables do not need to be declared and their data types are inferred from the assignment statement.
- Python supports the following data types.

PYTHON PROGRAMMING

DATATYPE	EXAMPLES
boolean integer long float string list object	bool = True age = 26 age = 65247852 pi = 3.14159 name = "Craig" print(name + ' is ' + str(age) + ' years old.') -> Craig is 26 years old.

VARIABLE ASSIGNMENT

- We use the assignment operator (=) to assign values to a variable.
- Any type of value can be assigned to any valid variable.

Example:

a = 5

b = 3.2

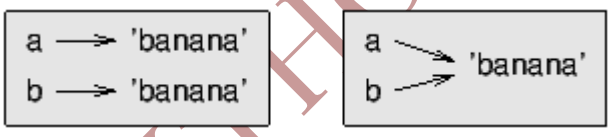
c = "Hello"

OBJECTS AND VALUES

If we execute these assignment statements:

a = 'banana'
b = 'banana'. Here a and b are objects."banana" is the value.

We know that a and b both refer to a string, but we don't know whether they refer to the *same* string. There are two possible states:



In one case, a and b refer to two different objects that have the same value. In the second case, they refer to the same object and separate memory is not allocated for each variable in python.

MULTIPLE ASSIGNMENTS

- In Python, multiple assignments can be made in a single statement as follows:

  Eg: a, b, c = 5, 3.2, "Hello"

- If we want to assign the same value to multiple variables at once, we can assign by the below mentioned way.

PYTHON PROGRAMMING

Eg: x = y = z = "same".This assigns the "same" string to all the three variables.

DELETE VARIABLE

- We can also delete variable using the command **del** .

Syntax: del "variable name"

Example: f=101;
print f
del f

Explanation:

When the above code,we deleted variable f and when we proceed to print it, we get error "**variable name is not defined**" which means we have deleted the variable.

PYTHON KEYWORDS AND IDENTIFIER

KEYWORDS

- Keywords are the reserved words in Python.
- We cannot use a keyword as variable name, function name or any other identifier.
- They are used to define the syntax and structure of the Python language.
- There are 33 keywords and they are case sensitive.
- All the keywords except True, False and None are in lowercase and they must be written as it is. The list of all the keywords is given below.

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

IDENTIFIERS

- Identifier is the name given to entities like class, functions, variables etc. in Python.
- It helps differentiating one entity from another.

PYTHON PROGRAMMING

RULES FOR WRITING IDENTIFIERS

1. Identifiers can be a combination of letters in lowercase (a to z) or uppercase (A to Z) or digits (0 to 9) or an underscore (_).
   Eg: myClass, var_1 .
2. An identifier cannot start with a digit.
   Eg:1variable is invalid, but variable1 is perfectly fine.
3. Keywords cannot be used as identifiers.

```
>>> global = 1
  File "<interactive input>", line 1
    global = 1
           ^
Syntax Error: invalid syntax
```

4. We cannot use special symbols like !, @, #, $, % etc.

```
>>> a@ = 0
  File "<interactive input>", line 1
    a@ = 0
     ^
SyntaxError: invalid syntax
```

5. Identifier can be of any length.

Python Comments

Python supports two types of comments:

1) Single lined comment:

In case user wants to specify a single line comment, then comment must start with ?#?

Eg:# This is single line comment.

2) Multi lined Comment:

Multi lined comment can be given inside triple quotes.

eg:

""" This

Is

Multipline comment"'

eg:

#single line comment

PYTHON PROGRAMMING

print "Hello Python"

"""This is

multiline comment"'

ASKING THE USER FOR INPUT

Python provides a built-in function called input that gets input from the keyboard. When this function is called, the program stops and waits for the user to type something. When the user presses Return or Enter , the program resumes and input returns what the user typed as a string.

>>> name = input()

Some silly stuff

The function input () is used to get the input from the user.

Eg 1:

Name=input("enter the value")

Eval() function is used to convert the given input as of the same data type given by the user

Eg 2:

Name=eval(input("enter the value"))

Enter the value:123

Here 123 is considered as integer itself.

Examples:

1.name=input("Enter your name")

print("Welcome", name)

2.name1=str(input("Enter your name"))

print("Welcome", name1)

3.Value1=int(input("enter the number"))

Print("The number is",value1)

4.Value2=float((input("enter the number"))

Print("The number is",value2)

5. Value3=complex(input("enter the number"))

Print("The number",value3).

PYTHON PROGRAMMING

CONVERSION BETWEEN DATA TYPES

- We can convert between different data types by using type conversion functions like int(), float(), str() etc.

  Example:
  >>> float(5)
  5.0

- Conversion from float to int will truncate the value (make it closer to zero).

  Example: >>> int(10.6)
  10
  Example: >>> int(-10.6)
  -10

- Conversion to and from string must contain compatible values.

  Example: >>> float('2.5')
  2.5
  Example: >>> str(25)
  '25'

- We can even convert one sequence to another.

  Example: >>> set([1,2,3])
  {1, 2, 3}
  Example: >>> tuple({5,6,7})
  (5, 6, 7)
  Example: >>> list('hello')
  ['h', 'e', 'l', 'l', 'o']

- To convert to dictionary, each element must be a pair

  Example: >>> dict([[1,2],[3,4]])
  {1: 2, 3: 4}
  Example: >>> dict([(3,26),(4,44)])
  {3: 26, 4: 44}

STATEMENTS

A statement is a unit of code that the Python interpreter can execute. We have seen two kinds of statements:

1. **print**
2. **Assignment**

PYTHON PROGRAMMING

When you type a statement in interactive mode, the interpreter executes it and displays the result, if there is one. A script usually contains a sequence of statements. If there is more than one statement, the results appear one at a time as the statements execute.

Example:

```
print 1

x = 2

print x
```

Output:

```
1

2
```

The assignment statement produces no output.

EXPRESSIONS

An expression is a combination of values, variables, and operators. A value all by itself is considered an expression, and so is a variable, so the following are all legal expressions (assuming that the variable x has been assigned a value):

```
17

x

x + 17
```

If you type an expression in interactive mode, the interpreter evaluates it and displays the result:

```
>>> 1 + 1

2
```

VALUES AND TYPES

A value is one of the basic things a program works with, like a letter or a number.The values we have seen so far are 1, 2, and 'Hello, World!'.These values belong to different types: 2 is an integer, and 'Hello, World!' is a string, so called because it contains a “string” of letters.

```
>>> print 4

4
```

If you are not sure what type a value has, the interpreter can tell you.

Example 1

```
>>> type('Hello, World!')
```

PYTHON PROGRAMMING

<type 'str'>

Example 2:

>>> type(17)

<type 'int'>

Not surprisingly, strings belong to the type str and integers belong to the type int. Less obviously, numbers with a decimal point belong to a type called float, because these numbers are represented in a format called floating point.

Example 3

>>> type(3.2)

<type 'float'>

id() function:

This function is used to find the memory address of the variable.

Eg:

Val1=10

id(Val1)

OUTPUT:12456677788

FORMAT OPERATOR

The **format operator**, % allows us to construct strings, replacing parts of the strings with the data stored in variables. When applied to integers, % is the modulus operator. But when the first operand is a string, % is the format operator. The first operand is the **format string**, which contains one or more **format sequences** that specify how the second operand is formatted. The result is a string. For example, the format sequence '%d' means that the second operand should be formatted as an integer (d stands for "decimal").

Example 1:

```
>>> camels = 42
>>> '%d' % camels
'42'
```

The result is the string '42', which is not to be confused with the integer value 42.A format sequence can appear anywhere in the string, so we can embed a value in a sentence:

Example 2:

```
>>> camels = 42
>>> 'I have spotted %d camels.' % camels
```

PYTHON PROGRAMMING

'I have spotted 42 camels.'

>>>>x,y=4,10

print("{} and {}".format(x,y))

4 and 10

Example 3:

```
>>> format(3.145555,"<35")
'3.145555                          '
>>> print(3.1444,">35.2")
3.1444 >35.2
>>> format(3.1444,">35.2")
'                                3.1'
>>> format(35.1444,">35.2")
'                             3.5e+01'
>>> format(35.1444,">35.22")
'            35.144399999999741931'
>>> format(35.1444,">35.2")
'                             3.5e+01'
>>> format(35.1444,">35.2f")
'                               35.14'
>>> format(35.1444,"^35.2")
'               3.5e+01               '
>>> format(35.144,"^35.2")
'               3.5e+01               '
>>> format(35.144,"^35.4")
'                35.14                '
>>> format(35.1444,"@^35.22")
'@@@@@@35.144399999999741931@@@@@@'
```

OPERATORS AND OPERANDS

- Operators are special symbols that represent computations like addition and multiplication.
- The values the operator is applied to are called operands.
- The operators +, -, *, /, and ** perform addition, subtraction, multiplication,division, and exponentiation

Examples:

20+32

hour-1

hour*60+minute

minute/60

PYTHON

5**2

(5+9)*(15-7)

The division operator might not do what you expect:

>>> minute = 59

>>> minute/60

0

Explanation:

The value of minute is 59, and in conventional arithmetic 59 divided by 60 is 0.98333, not 0. The reason for the discrepancy is that Python is performing floor division 2 .When both of the operands are integers, the result is also an integer; floor division chops off the fractional part, so in this example it truncates the answer to zero.If either of the operands is a floating-point number, Python performs floating-point division, and the result is a float:

>>> minute/60.0

0.98333333333333328.

Python Operators:

1. Arithmetic Operators.
2. Relational Operators.
3. Assignment Operators.
4. Logical Operators.
5. Membership Operators.
6. Identity Operators.
7. Bitwise Operators

Arithmetic Operators:

Operators	Description
//	Perform Floor division(gives integer value after division)
+	To perform addition
-	To perform subtraction
*	To perform multiplication

PYTHON PROGRAMMING

/	To perform division
%	To return remainder after division(Modulus)
**	Perform exponent(raise to power)

Relational Operators:

Operators	Description
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
!=	Not equal to
<>	Not equal to(similar to !=)

Assignment Operators:

Operators	Description
=	Assignment
/=	Divide and Assign
+=	Add and assign
-=	Subtract and Assign
*=	Multiply and assign
%=	Modulus and assign
**=	Exponent and assign

PYTHON PROGRAMMING

//=	Floor division and assign

Logical Operators:

Operators	Description
and	Logical AND(When both conditions are true output will be true)
or	Logical OR (If any one condition is true output will be true)
not	Logical NOT(Compliment the condition i.e., reverse)

Eg 1:

```
a=5>4 and 3>2
print( a)
b=5>4 or 3<2
print (b )
c=not(5>4)
print (c)
```

output:

```
True
True
False
```

Special Operators

Membership Operators:

Operators	Description
in	Returns true if a variable is in sequence of another variable, else false.
not in	Returns true if a variable is not in sequence of another variable, else false.

Identity Operators:

PYTHON PROGRAMMING

Operators	Description
Is	Returns true if identity of two operands are same, else false .
is not	Returns true if identity of two operands are not same, else false.

Example:

```
a=20
b=20
if a is b:
   print (a,b have same identity)
else:
   print(a, b are different)
b=10
if  a is not b :
   print(a,b have different identity)
else:
   print(a,b have same identity)
```

ORDER OF OPERATIONS

When more than one operator appears in an expression, the order of evaluation depends on the rules of precedence. For mathematical operators, Python follows mathematical convention. The acronym **PEMDAS** is a useful way to remember the rules:

• **Parentheses** have the highest precedence and can be used to force an expression to evaluate. Since expressions in parentheses are evaluated first, 2 * (3-1) is 4, and (1+1)**(5-2) is 8. We also use parentheses to make an expression easier to read, as in (minute * 100) /60.

• **Exponentiation** has the next highest precedence, so 2**1+1 is 3, not 4, and 3*1**3 is 3, not 27.

• **Multiplication** and **Division** have the same precedence, which is higher than Addition and Subtraction, which also have the same precedence. So 2*3-1 is 5, not 4, and 6+4/2 is 8, not 5.

• Operators with the same precedence are evaluated from **left to right**. So the expression 5-3-1 is 1, not 3, because the 5-3 happens first and then 1 is subtracted from 2.

PYTHON PROGRAMMING

MODULUS OPERATOR

The modulus operator works on integers and yields the remainder when the first operand is divided by the second. The syntax is the same as for other operators

Example:

```
quotient = 7 / 3
print (quotient)
remainder = 7 % 3
print (remainder)
```

Built-in-functions in python

1. id()-It is used to find the memory address of the variable.

Eg:

Val1=10

id(Val1)

OUTPUT: 12456677788

2. **type()**-This function is used to find the type of the variable.

Val1=10

type (Val1)

OUTPUT: class_int

3. **getsizeof()**-This function is used to find the size of variable. We must import the package **import sys** to use this function

```
>>> import sys
>>> a=10
>>> sys.getsizeof(a)
14
>>> sys.getsizeof('hai')
28
>>> sys.getsizeof((1,))
32
>>> sys.getsizeof(1.34)
16
>>>
```

4. We can use math function by importing the package **import math.**

PYTHON PROGRAMMING

```
>>> math.sqrt(4)
2.0
>>> math.ceil(2.2)
3
>>> math.floor(2.3)
2
>>> math.floor(2.8)
2
>>> math.floor(-1.1)
-2
>>> val=20,30,40
>>> sum(val)
90
```

```
>>> round(3.1577,2)
3.16
>>> min(7,8)
7
>>> max(8,9)
9
```

5. We can use **decimal and fraction** function by importing the package **import Decimal/import Fractions.**

```
>>> import decimal
>>> decimal.Decimal(3.14)
Decimal('3.140000000000000124344978758017532527446746826171875')
>>>
==================== RESTART: C:/Users/Manju/Desktop/a.py =============
<module 'sys' (built-in)>
>>> import fraction
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    import fraction
ImportError: No module named 'fraction'
>>> import fractions
>>> fractions.Fraction(0.155)
Fraction(5584463537939415, 36028797018963968)
>>> print(fractions.Fraction(0.155))
5584463537939415/36028797018963968
```

6.**ord and chr**

Ord-It is used to find the ASCII value for the corresponding character.

chr-It is used to find the character for the ascii value

Example:

```
>>> ord('A')
65
>>> chr(65)
'A'
```

7. We can able to display all the keywords by importing keyword package

Example:
import keyword
>>> print(keyword.kwlist)

Output:
['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']
8. **sep** functions are used to separate the strings with the specified character and **end** function is used to identify the end of the string.
Example 1:
print("hai","hello",sep='$')
>>> hai$hello
print("hai","hello",end='$')
>>>hai hello$

CONDITIONAL EXECUTION

BOOLEAN EXPRESSIONS

A boolean expression is an expression that is either true or false. The following examples use the operator ==, which compares two operands and produces True if they are equal and False otherwise:

>>> 5 == 5

True

>>> 5 == 6

False

True and False are special values that belong to the type bool; they are not strings:

>>> type(True)

PYTHON PROGRAMMING

<type 'bool'>

>>> type(False)

<type 'bool'>

```
x != y               # x is not equal to y
x > y                # x is greater than y
x < y                # x is less than y
x >= y               # x is greater than or equal to y
x <= y               # x is less than or equal to y
x is y               # x is the same as y
x is not y           # x is not the same as y
```

Although these operations are probably familiar to you, the Python symbols are different from the mathematical symbols for the same operations. A common error is to use a single equal sign (=) instead of a double equal sign (==). Remember that = is an assignment operator and == is a comparison operator.

LOGICAL OPERATORS

There are three logical operators: and, or, and not. The semantics (meaning) of these operators is similar to their meaning in English. For example,

x > 0 and x < 10

is true only if x is greater than 0 and less than 10.

n%2 == 0 or n%3 == 0 is true if either of the conditions is true, that is, if the number is divisible by 2 or 3.

Finally, the not operator negates a boolean expression, so not (x > y) is true if x > y is false; that is, if x is less than or equal to y. Strictly speaking, the operands of the logical operators should be boolean expressions, but Python is not very strict. Any nonzero number is interpreted as "true."<<< 17 and True.

This flexibility can be useful, but there are some subtleties to it that might be confusing.

CONDITIONAL EXECUTION

If statement:

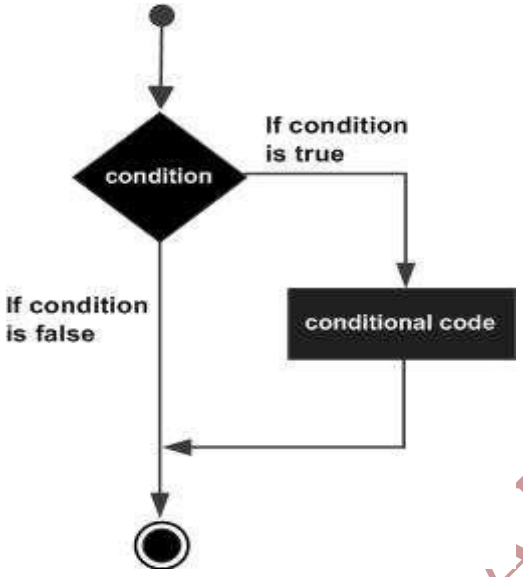
Syntax:

```
if(condition):
  statements
```

In order to write useful programs, we almost always need the ability to check conditions and change the behavior of the program accordingly. Conditional statements give us this ability. The simplest form is the if statement:

PYTHON PROGRAMMING

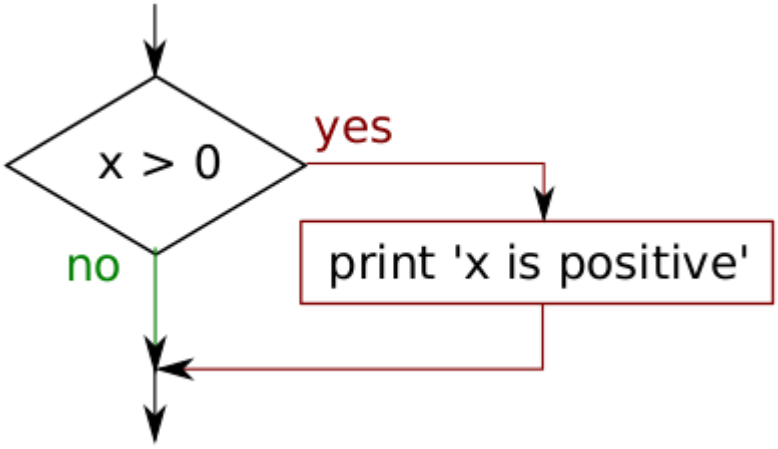
Flow diagram:



Example 1:

```
if x > 0 :
        print( 'x is positive')
```

The boolean expression after the if statement is called the condition. We end the if statement with a colon character (:) and the line(s) after the if statement are indented.



If the logical condition is true, then the indented statement gets executed. If the logical condition is false, the indented statement is skipped.

Example 2:

```
var1 = 100
if var1:
   print("1 - Got a true expression value")
```

PYTHON PROGRAMMING

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

Output:

```
1 - Got a true expression value
100
Good bye!
```

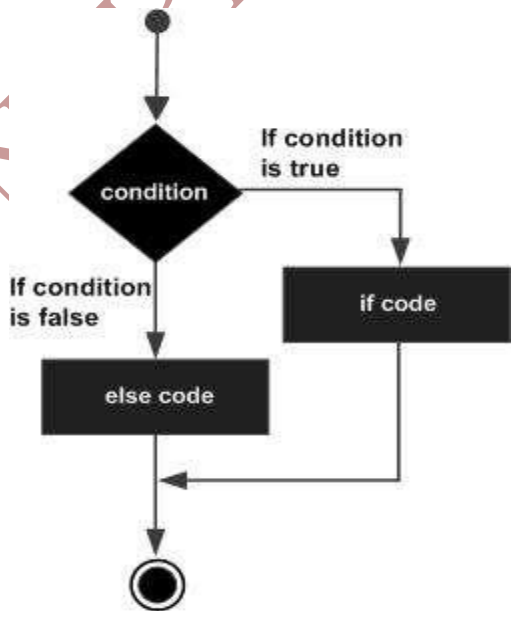
ALTERNATIVE EXECUTION

If else statement:
Syntax:

```
if(condition):
        statements
   else:
        statements
```

A second form of the if statement is **alternative execution**, in which there are two possibilities and the condition determines which one gets executed. The syntax looks like this:

Flow Diagram:



Example 1:

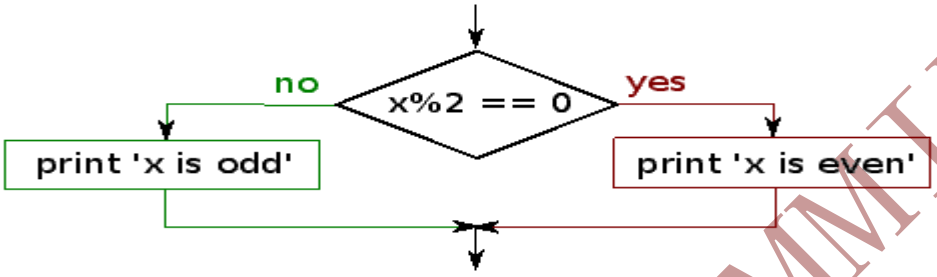
```
if x%2 == 0 :
   print( 'x is even')
```

PYTHON PROGRAMMING

```
else :
    print ('x is odd')
```

Explanation:

If the remainder when x is divided by 2 is 0, then we know that x is even, and the program displays a message to that effect. If the condition is false, the second set of statements is executed.



Since the condition must be true or false, exactly one of the alternatives will be executed. The alternatives are called **branches**, because they are branches in the flow of execution.

Example 2:

```
var1 = 100
if var1:
   print ("1 - Got a true expression value")
   print (var1)
else:
   print ("1 - Got a false expression value")
   print (var1)
var2 = 0
if var2:
   print( "2 - Got a true expression value")
   print (var2)

else:
  print( "2 - Got a false expression value")
  print (var2)
print ("Good bye!")
```

Output:

```
1 - Got a true expression value
100
2 - Got a false expression value
0
Good bye!
```

CHAINED CONDITIONALS

PYTHON PROGRAMMING

Sometimes there are more than two possibilities and we need more than two branches. One way to express a computation like that is a **chained conditional**:

Nested if:

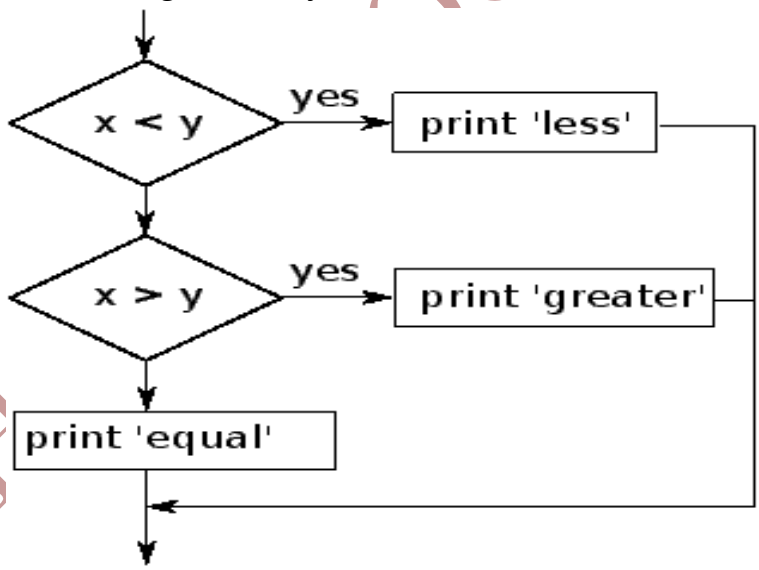
Syntax:

```
If statement:
    Body
elif statement:
    Body
else:
    Body
```

Example 1:

```
if x < y:
    print( 'x is less than y')
elif x > y:
    print( 'x is greater than y')
else:
    print( 'x and y are equal')
```

elif is an abbreviation of "else if." Again, exactly one branch will be executed.



There is no limit on the number of elif statements. If there is an else clause, it has to be at the end, but there doesn't have to be one.

Example 2:

```
if choice == 'a':
    print ('Bad guess')
elif choice == 'b':
```

PYTHON PROGRAMMING

```
    print ('Good guess')
elif choice == 'c':
    print('Close, but not correct')
```

Each condition is checked in order. If the first is false, the next is checked, and so on. If one of them is true, the corresponding branch executes, and the statement ends. Even if more than one condition is true, only the first true branch executes.

Example 3:

```
var = 100
if var == 200:
   print("1 - Got a true expression value")
   print (var)
elif var == 150:
   print ("2 - Got a true expression value")
   print (var)
elif var == 100:
   print ("3 - Got a true expression value")
    print (var)
else:
   print ("4 - Got a false expression value")
   print (var)
print ("Good bye!")
```

Output:

```
3 - Got a true expression value
100
Good bye!
```

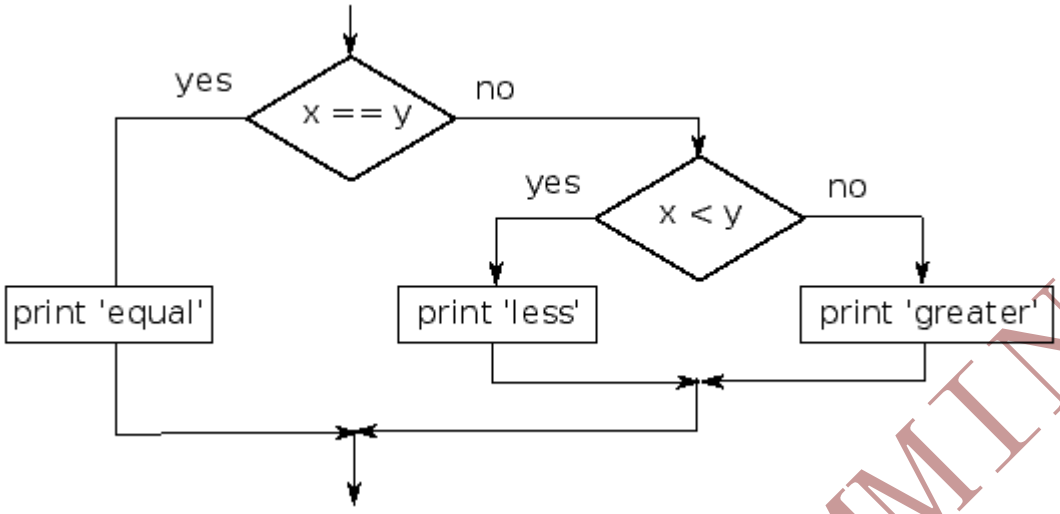
NESTED CONDITIONALS

One conditional can also be nested within another. We could have written the example like this:

```
if x == y:
    print ('x and y are equal')
else:
    if x < y:
        print ('x is less than y')
    else:
        print ('x is greater than y')
```

The outer conditional contains two branches. The first branch contains a simple statement. The second branch contains another if statement, which has two branches of its own. Those two branches are both simple statements, although they could have been conditional statements as well.

PYTHON PROGRAMMING



Although the indentation of the statements makes the structure apparent, **nested conditionals** become difficult to read very quickly. Logical operators often provide a way to simplify nested conditional statements. For example, we can rewrite the following code using a single conditional:

```
if 0 < x:
   if x < 10:
      print ('x is a positive single-digit number.')
```

The print statement is executed only if we make it past both conditionals, so we can get the same effect with the and operator:

```
if 0 < x and x < 10:
   print( 'x is a positive single-digit number.')
```

SHORT CIRCUIT EVALUATION OF LOGICAL EXPRESSIONS

When Python is processing a logical expression such as x >= 2 and (x/y) > 2, it evaluates the expression from left-to-right. Because of the definition of and, if x is less than 2, the expression x >= 2 is False and so the whole expression is False regardless of whether (x/y) > 2 evaluates to True or False.

When Python detects that there is nothing to be gained by evaluating the rest of a logical expression, it stops its evaluation and does not do the computations in the rest of the logical expression. When the evaluation of a logical expression stops because the overall value is already known, it is called **short-circuiting** the evaluation.

While this may seem like a fine point, the short circuit behavior leads to a clever technique called the **guardian pattern**. Consider the following code sequence in the Python interpreter:

Examples:

```
>>> x = 6
>>> y = 2
>>> x >= 2 and (x/y) > 2
```

PYTHON PROGRAMMING

```
True
>>> x = 1
>>> y = 0
>>> x >= 2 and (x/y) > 2
False
>>> x = 6
>>> y = 0
>>> x >= 2 and (x/y) > 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
>>>
```

The third calculation failed because Python was evaluating (x/y) and y was zero which causes a runtime error. But the second example did *not* fail because the first part of the expression x >= 2 evaluated to False so the (x/y) was not ever executed due to the **short circuit** rule and there was no error. We can construct the logical expression to strategically place a **guard** evaluation just before the evaluation that might cause an error as follows:

```
>>> x = 1
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False
>>> x = 6
>>> y = 0
>>> x >= 2 and y != 0 and (x/y) > 2
False
>>> x >= 2 and (x/y) > 2 and y != 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
>>>
```

In the first logical expression, x >= 2 is False so the evaluation stops at the end. In the second logical expression x >= 2 is True but y!= 0 is False so we never reach (x/y). In the third logical expression, the!= 0 is *after* the (x/y) calculation so the expression fails with an error. In the second expression, we say that y != 0 acts as a **guard** to insure that we only execute (x/y) if y is non-zero.

Looping statements:

1. for loop
2. while loop

For loop:

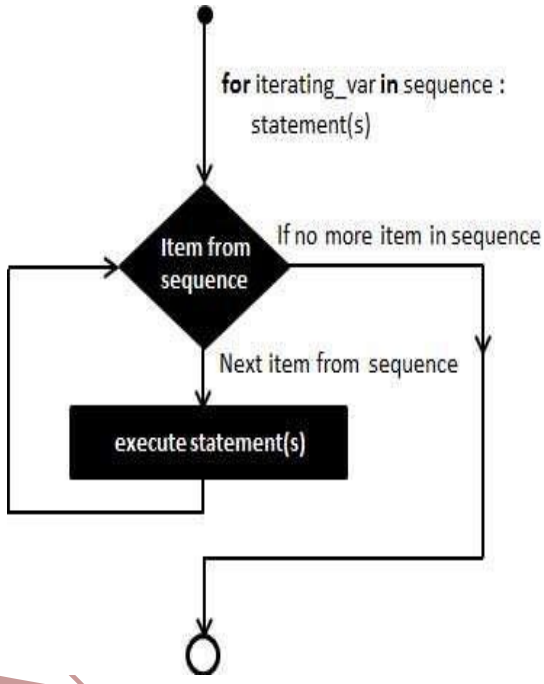
PYTHON PROGRAMMING

Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable. It has the ability to iterate over the items of any sequence, such as a list or a string.

Syntax:

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

Flow Diagram:



Example 1:

```
for letter in 'Python':
    print ('Current Letter :', letter)
fruits = ['banana', 'apple', 'mango']

for fruit in fruits:
   print ('Current fruit :', fruit)
print ("Good bye!")
```

Output:

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

PYTHON PROGRAMMING

```
Current fruit : banana
Current fruit : apple
Current fruit : mango
Good bye!
```

Iterating by Sequence Index

An alternative way of iterating through each item is by index offset into the sequence itself.

Example 1:

```
fruits = ['banana', 'apple', 'mango']
for index in range(len(fruits)):
    print( 'Current fruit :', fruits[index])
print("Good bye!")
```

Output:

```
Current fruit : banana
Current fruit : apple
Current fruit : mango
Good bye!
```

Using else Statement with Loops

Example 1:

```
for num in range(10,20):                    #to iterate between 10 to 20
     for i in range(2,num):                      #to iterate on the factors of the number
       if num%i == 0:                          #to determine the first factor
       j=num/i                                 #to calculate the second factor
      print ('%d equals %d * %d' % (num,i,j))
      break                                  #to move to the next number, the #first FOR
else:                                         # else part of the loop
     print( num, 'is a prime number')
```

Output:

```
10 equals 2 * 5
11 is a prime number
12 equals 2 * 6
13 is a prime number
14 equals 2 * 7
15 equals 3 * 5
16 equals 2 * 8
17 is a prime number
18 equals 2 * 9
19 is a prime number
```

Example 2:

```
num=2
```

PYTHON PROGRAMMING

```
for a in range (1,6):

print ( num * a)
```

Nested for Loops:

Loops defined within another Loop is called Nested Loop.

Syntax for nested for loop:

```
For iterating_var in sequence:
    for iterating_var in sequence:
      statements(s)
    statements(s)
```

Syntax for nested while loop:

```
while expression:
    while expression:
      statement(s)
    statement(s)
```

Example 1:

```
for i in range(1,6):
    for j in range (1,i+1):
        print( i,)
    print ()
```

Output:

```
1
2 2
3 3 3
4 4 4 4
5 5 5 5 5
```

Example 2:

Program to print Pyramid:

```
for i in range (1,6):
    for j in range (5,i-1,-1):
        print ("*",)
    print()
```

PYTHON PROGRAMMING

Output:

```
* * * * *

* * * *

* * *

* *

*
```

Range functions in loops:

It is used to iterate over the specific number of times within the given range in steps/intervals.

Syntax: range(lowerlimit,upperlimit,increment/decrement)

Example programs:

```
>>> for i in 1,2,3,4.5,2+3j:
        print(i)


1
2
3
4.5
(2+3j)
>>> for i in "hai":
        print(i)


h
a
i
>>> for i in range(1,6):
        print(i)


1
2
3
4
5
>>> for i in range(1,6,2):
        print(i)
```

PYTHON PROGRAMMING

```
>>> for i in range(1,6,2):
        print(i)


1
3
5
>>> for i in range(6,1,-2):
        print(i)


6
4
2
```

While loop:

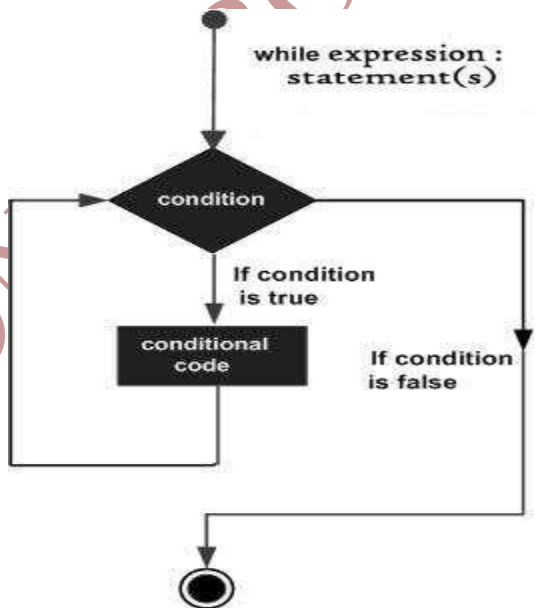
Repeats a statement or group of statements while the given condition is TRUE.

Syntax:

while (condition):

statements

Flow Diagram:



Example 1:

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

PYTHON PROGRAMMING

Output:

```
The count is: 0
The count is: 1
The count is: 2
The count is: 3
The count is: 4
The count is: 5
The count is: 6
The count is: 7
The count is: 8
Good bye!
```

The Infinite Loop

A loop becomes infinite loop if a condition never becomes FALSE. We must use caution when using while loops because of the possibility that this condition never resolves to a FALSE value. This results in a loop that never ends. Such a loop is called an infinite loop.

Example 1:

```
var = 1
while var == 1 :                    # This constructs an infinite loop
   num = input("Enter a number :")
   print ("You entered: ", num)
print ("Good bye!")
```

Output:

```
Enter a number: 20
You entered: 20
Enter a number: 29
You entered: 29
Enter a number: 3
You entered: 3
```

Using else Statement with Loops

Python supports to have an **else** statement associated with a loop statement.

1. If the **else** statement is used with a **for** loop, the **else** statement is executed when the loop has exhausted iterating the list.
2. If the **else** statement is used with a **while** loop, the **else** statement is executed when the condition becomes false.

Example 1:

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

PYTHON PROGRAMMING

Output:

0 is less than 5
1 is less than 5
2 is less than 5
3 is less than 5
4 is less than 5
5 is not less than 5

Loop Control Statements:

Break:

Terminates the loop statement and transfers execution to the statement immediately following the loop. The **break** statement can be used in both *while* and *for* loops.

Syntax: break

Example 1:

```
   for letter in 'Python':
        if letter == 'h':
            break
        print ('Current Letter :', letter)
var = 10
     while var > 0:
        print( 'Current variable value :', var)
        var = var -1
         if var == 5:
              break
    print ("Good bye!")
```

Output:

Current Letter : P
Current Letter : y
Current Letter : t
Current variable value : 10
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
Good bye!

Continue:

It returns the control to the beginning of the while loop. The **continue** statement rejects all the remaining statements in the current iteration of the loop and moves the control back to the top of the loop. The **continue** statement can be used in both *while* and *for* loops.

Syntax: continue

PYTHON PROGRAMMING

Example 1:

```
for letter in 'Python':
    if letter == 'h':
        continue
print('Current Letter :', letter)
var = 10
while var > 0:
    var = var -1
    if var == 5:
        continue
    print('Current variable value :', var)
print ("Good bye!")
```

Output:

```
Current Letter : n
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
Current variable value : 4
Current variable value : 3
Current variable value : 2
Current variable value : 1
Current variable value : 0
Good bye!
```

Pass:

The **pass** statement is a *null* operation; nothing happens when it executes.

Syntax:pass

Example 1:

```
for letter in 'Python':
     if letter == 'h':
       pass
       print('This is pass block')
      print ('Current Letter :', letter)
print ("Good bye!")
```

Output:

```
Current Letter : P
Current Letter : y
Current Letter : t
This is pass block
Current Letter : h
Current Letter : o
Current Letter : n
Good bye!
```

PYTHON PROGRAMMING

DATA TYPES IN PYTHON

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

1. Numbers
2. Strings
3. List
4. Tuple
5. Set
6. Dictionary

1. PYTHON NUMBERS

Python's built-in core data types are in some cases also called object types. There are four built-in data types for numbers:

- **Integer**
  - **Normal integers**
    e.g. 4321
  - **Octal literals (base 8)**
    A number prefixed by a 0 (zero) will be interpreted as an octal number
    Example:
    >>> a = 010
    >>> print a
    8 Alternatively, an octal number can be defined with "0o" as a prefix: >>> a = o10
    >>> print a
    8
  - **Hexadecimal literals (base 16)**
    Hexadecimal literals have to be prefixed either by "0x" or "0X".
    Example:
    >>> hex_number = 0xA0F
    >>> print hex_number
    2575
  - **Long integers**
    These numbers are of unlimited size.
    e.g.42000000000000000000L
  - **Floating-point numbers**
    Example: 42.11, 3.1415e-10

PYTHON PROGRAMMING

- **Complex numbers**

  Complex numbers are written as <*real part*> + <*imaginary part*>j

  Examples:

  >>> x = 3 + 4j

  >>> y = 2 - 3j

  >>> z = x + y

  >>> print z

  (5+1j)

2. STRINGS

- Wrapped with the single-quote (') character-'This is a string with single quotes' .
- Wrapped with the double-quote (") character-"Obama's dog is called Bo"
- Wrapped with three characters, using either single-quote or double-quote: '''A String in triple quotes can extend over multiple lines like this one, and can contain 'single' and "double" quotes.'''

- A string in Python consists of a series or sequence of characters - letters, numbers, and special characters.

- Strings can be indexed - often called subscript. The first character of a string has the index 0.

  Example 1:

  >>> s = "A string consists of characters"

  >>> s[0]

  'A'

  >>> s[3]

  't'.

- The last character of a string can be accessed by below mentioned way

  >>> s[len(s)-1]

  's'

- There is an easier way to access last character with -1, the second to last with -2 and so on:

  >>> s[-1]

  's'

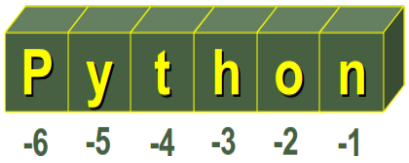
  >>> s[-2]

  'r'

- There exists no character type in Python. A character is simply a string of size 1byte.

PYTHON PROGRAMMING

- It's possible to start counting the indices from the right. In this case negative numbers are used, starting with -1 for the most right character.



ESCAPE SEQUENCES

- The backslash (\) character is used to escape characters.
- String literals may optionally be prefixed with a letter 'r' or 'R' these strings are called **raw strings**.

ESCAPE SEQUENCE	MEANING NOTES
\newline	Ignored
\\	Backslash (\)
\'	Single quote (')
\"	Double quote (")
\a	ASCII Bell (BEL)
\b	ASCII Backspace (BS)
\f	ASCII Formfeed (FF)
\n	ASCII Linefeed (LF)
\N{name}	Character named name in the Unicode database (Unicode only)
\r	ASCII Carriage Return (CR)
\t	ASCII Horizontal Tab (TAB)
\uxxxx	Character with 16-bit hex value xxxx (Unicode only)
\Uxxxxxxxx	Character with 32-bit hex value xxxxxxxx (Unicode only)
\v	ASCII Vertical Tab (VT)
\ooo	Character with octal value ooo
\xhh	Character with hex value hh

3. LIST

PYTHON PROGRAMMING

- List_is an ordered sequence of items. It is one of the most used datatype in Python and is very flexible. All the items in a list need not be of the same type.
- Declaring a list: Items separated by commas are enclosed within brackets [ ].

  Example:>>> a = [1, 2.2, 'python']
- We can use the slicing operator [ ] to extract an item or a range of items from a list.
- Index starts form 0 in Python.

  Example 1:

```
a = [5,10,15,20,25,30,35,40]
# a[2] = 15
print("a[2] = ", a[2])
# a[0:3] = [5, 10, 15]
print("a[0:3] = ", a[0:3])

# a[5:] = [30, 35, 40]
print("a[5:] = ", a[5:])
```

- Lists are mutable(i.e)value of elements of a list can be altered.

```
>>> a = [1,2,3]
>>> a[2]=4
>>> a
[1, 2, 4]
```

4. TUPLE

- Tuple is an ordered sequence of items same as list. The only difference is that Tuples are **immutable.** Tuples once **created cannot be modified.**
- Tuples are used to write-protect data and are usually faster than list as it cannot change dynamically.
- It is defined within parentheses () where items are separated by commas.

  Ex:>>> t = (5,'program', 1+3j).
- We can use the slicing operator [] to extract items but we cannot change its value.

  Example 1:

```
t = (5,'program', 1+3j)
```

PYTHON PROGRAMMING

```
# t[1] = 'program'

print("t[1] = ", t[1])

# t[0:3] = (5, 'program', (1+3j))

print("t[0:3] = ", t[0:3])

# Generates error
```

- Tuples are immutable

```
t[0] = 10
```

5. SET

- **Set** is an un-ordered collection of unique items. Set is defined by values separated by comma inside braces { }. Items in a set are not ordered.

Example 1:

```
= {5,2,3,1,4}
```

Example 2:

```
# printing set variable

print("a = ", a)
```

Example 3:

```
# data type of variable a

print(type(a))
```

- We can perform set operations like union, intersection on two sets. Set have unique values. They eliminate duplicates.

```
Example:>>> a = {1,2,2,3,3,3}
>>> a
{1, 2, 3}
```

- Since, set are unordered collection, indexing has no meaning. Hence the slicing operator [] does not work.

```
Example:>>> a = {1,2,3}
>>> a[1]
Traceback (most recent call last):
  File "<string>", line 301, in runcode
  File "<interactive input>", line 1, in <module>
TypeError: 'set' object does not support indexing
```

PYTHON PROGRAMMING

6. DICTIONARY

- Dictionary is an un-ordered collection of key-value pairs.
- It is generally used when we have a huge amount of data.
- Dictionaries are optimized for retrieving data. We must know the key to retrieve the value.
- Dictionaries are defined within braces {} with each item being a pair in the form key:value. Key and value can be of any type.

  Example:

  >>> d = {1:'value','key':2}

  >>> type(d)

  <class 'dict'>

- We use key to retrieve the respective value. But not the other way around.

  Example:

  d = {1:'value','key':2}

  print(type(d))

  print("d[1] = ", d[1]);

  print("d['key'] = ", d['key']);

  # Generates error

  print("d[2] = ", d[2]);

STRINGS

A STRING IS A SEQUENCE

- A string is a **sequence** of characters. It is immutable
- It can be enclosed in single quotes and double quotes.
- We use access the string by using indexing operator[]

>>> fruit = 'banana'

>>> letter = fruit[1]

The second statement extracts the character at index position 1 from the fruit variable and assigns it to letter variable. The expression in brackets is called an **index**. The index indicates which character in the sequence you want (hence the name).

>>> print (letter)

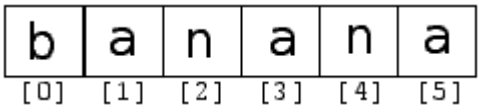
PYTHON PROGRAMMING

>>>a

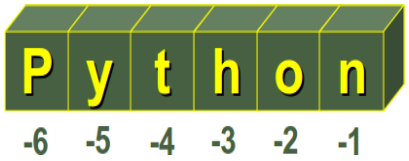
Python, the index is an offset from the beginning of the string, and the offset of the first letter is zero.

```
>>> letter = fruit[0]
>>> print letter
b
```

So b is the 0th letter ("zero-eth") of 'banana', a is the 1th letter ("one-eth"), and n is the 2th ("two-eth") letter.



It's possible to start counting the indices from the right. In this case negative numbers are used, starting with -1 for the most right character.



STRINGS ARE IMMUTABLE

It is tempting to use the [ ] operator on the left side of an assignment, with the intention of changing a character in a string.

Example:

```
>>> greeting = 'Hello, world!'
>>> greeting[0] = 'J'
TypeError: object does not support item assignment
```

The "object" in this case is the string and the "item" is the character you tried to assign. For now, an **object** is the same thing as a value, but we will refine that definition later. An **item** is one of the values in a sequence.The reason for the error is that strings are **immutable**, which means you can't change an existing string. The best you can do is creating a new string that is a variation on the original.

```
>>> greeting = 'Hello, world!'
>>> new_greeting = 'J' + greeting[1:]
>>> print new_greeting
Jello, world!
```

Whenever you try to modify an existing string variable, a new string will be created. Every object in python is stored in memory.by using id ( ) function it can be detected whether two variables are referring to the same object or not.

PYTHON PROGRAMMING

Example:

```
Str1="all is well"

Str2="be happy"

print("address is :", id(Str1),id(Str2))

Str1+=Str2

print(id(Str1))

Str3=Str1

print(id(Str3))
```

Output:

```
address is : 60150152 60150312

60144528

60144528
```

From the output it is very clear that str1 an str2 are two different string objects with different values and have a different memory address. When string concatenation is done with str1 and str2 new string is created because the strings are immutable in nature.

STRING OPERATIONS

1. SLICING

2. STRIDING

Slicing:

We can slice the particular character/substring by using slicing operator [n.m].

n-It is the starting value of string and it is included.

m-It is the ending value of string and it is excluded.

```
>>>>name=”python”

>>>name[0:3]

>>>pyth
```

PYTHON PROGRAMMING

Examples:

```
>>> s="python ptogramming"
>>> s[0:6]
'python'
>>> s[:9]
'python pt'
>>> s[-6:]
'amming'
>>> s[3:9]
'hon pt'
>>> s[-6:-1]
'ammin'
>>>
```

Striding:

The optional, third number in a slice specifies the *stride*. If omitted, the default is 1: return every character in the requested range. To return every kth letter, set the stride to k. Negative values of k reverse the string.

Syntax:[start size:end size:step size]

Example1 :

```
>>> s="python"
>>> s[::-1]
'nohtyp'

>>> s[::1]
'python'

>>> s[1::]
'ython'

>>> s[::2]
'pto'

>>> s[:3:]
'pyt'

>>> s[::-1]
'nohtyp'

>>> s[-6:-1:2]
'pto'

>>> s[-4::-1]
'typ'
```

PYTHON PROGRAMMING

Example 2:

```
>>> s="welcome"
>>> s[-4:-1]
'com'

>>> s[-4:-1:1]
'com'
```

String Special Operators:

There are basically 3 types of Operators supported by String:

1. Basic Operators.
2. Membership Operators.
3. Relational Operators.

Example 1:

A=”hello”

B=”python”

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

GETTING THE LENGTH OF A STRING USING LEN( )

len is a built-in function that returns the number of characters in a string:

```
>>> fruit = 'banana'
>>> len(fruit)
6
```

To get the last letter of a string, you might be tempted to try something like this:

```
>>> length = len(fruit)
>>> last = fruit[length]
IndexError: string index out of range
```

The reason for the Index Error is that there is no letter in 'banana' with the index 6. Since we started counting at zero, the six letters are numbered 0 to 5. To get the last character, you have to subtract 1 from length:

```
>>> last = fruit[length-1]
>>> print last
a
```

Alternatively, you can use negative indices, which count backward from the end of the string. The expression fruit[-1] yields the last letter, fruit[-2] yields the second to last, and so on.

TRAVERSAL THROUGH A STRING WITH A LOOP

A lot of computations involve processing a string one character at a time. Often they start at the beginning, select each character in turn, do something to it, and continue until the end. This pattern of processing is called a **traversal**. One way to write a traversal is with a while loop:

```
index = 0
while index < len(fruit):
    letter = fruit[index]
    print letter
    index = index + 1
```

This loop traverses the string and displays each letter on a line by itself. The loop condition is index < len(fruit), so when index is equal to the length of the string, the condition is false, and the body of the loop is not executed. The last character accessed is the one with the index len(fruit)-1, which is the last character in the string.

LOOPING AND COUNTING

The following program counts the number of times the letter a appears in a string:

```
word = 'banana'
count = 0
for letter in word:
    if letter == 'a':
        count = count + 1
            print count
```

PYTHON PROGRAMMING

This program demonstrates another pattern of computation called a **counter**. The variable count is initialized to 0 and then incremented each time an a is found. When the loop exits, count contains the result---the total number of a's.

1
2
3

THE IN OPERATOR

The word in is a boolean operator that takes two strings and returns True if the first appears as a substring in the second:

```
>>> 'a' in 'banana'
True
>>> 'seed' in 'banana'
False
```

STRING METHODS

Strings are an example of Python objects. An object contains both data (the actual string itself) as well as methods, which are effectively functions which that are built into the object and are available to any instance of the object.

Python has a function called dir that lists the methods available for an object. The type function shows the type of an object and the dir function shows the available methods.

```
>>> stuff = 'Hello world'
>>> type(stuff)
<type 'str'>

>>> dir(stuff)
```

['capitalize', 'center', 'count', 'decode', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'index', 'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']

Built-in Functions in Python

capitalize()	It capitalizes the first character of the String.
count(string,begin,end)	Counts number of times substring occurs in a String between begin and end index.
endswith(suffix	Returns a Boolean value if the string terminates with

PYTHON PROGRAMMING

,begin=0,end=n)	given suffix between begin and end.
find(substring ,beginIndex, endIndex)	It returns the index value of the string where substring is found between begin index and end index.
index(subsring, beginIndex, endIndex)	Same as find() except it raises an exception if string is not found.
isalnum()	It returns True if characters in the string are alphanumeric i.e., alphabets or numbers and there is at least 1 character. Otherwise it returns False.
isalpha()	It returns True when all the characters are alphabets and there is at least one character, otherwise False.
isdigit()	It returns True if all the characters are digit and there is at least one character, otherwise False.
islower()	It returns True if the characters of a string are in lower case, otherwise False.
isupper()	It returns False if characters of a string are in Upper case, otherwise False.
isspace()	It returns True if the characters of a string are whitespace, otherwise false.
len(string)	len() returns the length of a string.
lower()	Converts all the characters of a string to Lower case.
upper()	Converts all the characters of a string to Upper Case.
startswith(str ,begin=0,end=n)	Returns a Boolean value if the string starts with given str between begin and end.
swapcase()	Inverts case of all characters in a string.

lstrip()	Remove all leading whitespace of a string. It can also be used to remove particular character from leading.
rstrip()	Remove all trailing whitespace of a string. It can also be used to remove particular character from trailing.

Example 1:

```
s="Java Programming"
print("captialize",s.capitalize())
print("center",s.center(60,'*'))

ss="o"

sss="Java"
print("count",s.count(ss))
print("count",s.count('rm',3,10))
print("find",s.find(sss))
print(len(s))

a="Alpha123"
print("alphanumeric",a.isalnum())
print("Alphabets",a.isalpha())

a1="12345"
print("Digits",a1.isdigit())
print("Space",a1.isspace())
print("isDecimal",a1.isdecimal())
print("Maximum value",max(s))
print("Minimum value",min(s))
print(s.replace("Java", "python"))
print("split",s.split())
print("lower",s.lower())
print("upper",s.upper())
print(s)
```

PYTHON PROGRAMMING

Output:

```
captialize Java programming
center **********Java Programming**********
count 1
count 0
find 0
16
alphanumeric True
Alphabets False
Digits True
Space False
isDecimal True
Maximum value v
Minimum value
python Programming
split ['Java', 'Programming']
lower java programming
upper JAVA PROGRAMMING
Java Programming
```

Sample program:

capitalize ()

```
str = "this is string example....wow!!!"
print ("str.capitalize() : ", str.capitalize())
```

Result

When we run above program, it produces the following result —

```
str.capitalize() :  This is string example....wow!!!
```

endswith()

```
Str='this is string example....wow!!!'
suffix='!!'
print (Str.endswith(suffix))
print (Str.endswith(suffix,20))
suffix='exam'
print (Str.endswith(suffix))
print (Str.endswith(suffix, 0, 19))
```

Result

```
True
True
False
True
```

Join()

```
s = "-"
seq = ("a", "b", "c") # This is sequence of strings.
print (s.join( seq ))
```

Result

When we run above program, it produces the following result

```
a-b-c
```

Lstrip()

```
str = "     this is string example....wow!!!"
print (str.lstrip())

str = "*****this is string example....wow!!!*****"
print (str.lstrip('*'))
```

When we run above program, it produces the following result –

```
this is string example....wow!!!
this is string example....wow!!!*****
```

Replace()

```
str = "this is string example....wow!!! this is really string"
print (str.replace("is", "was"))
print (str.replace("is", "was", 3))
```

Result

When we run above program, it produces the following result

```
thwas was string example....wow!!! thwas was really string
thwas was string example....wow!!! thwas is really string
```

PYTHON PROGRAMMING

Example 6.15 Program to use slice operation with stride

```
str = "Welcome to the world of Python"
print("str[2:10] = ", str[2:10])    # default stride is 1
print("str[2:10:1] = ", str[2:10:1])   # same as stride = 1
print("str[2:10:2] = ", str[2:10:2])   # skips every alternate character
print("str[2:13:4] = ", str[2:13:4])   # skips every fourth character
```

OUTPUT

```
str[2:10] = lcome to
str[2:10:1] = lcome to
str[2:10:2] = loet
str[2:13:4] = le
```

String comparison operators:

String Comparison Operators and their

Operator	Description	Example
==	If two strings are equal, it returns True.	>>> "AbC" == "AbC" True

!= or <>	If two strings are not equal, it returns True.	>>> "AbC" != "Abc"
		True
		>>> "abc" <> "ABC"
		True

>	If the first string is greater than the second, it returns True.	>>> "abc" > "Abc" True
<	If the second string is greater than the first, it returns True.	>>> "abC" < "abc" True
>=	If the first string is greater than or equal to the second, it returns True.	>>> "aBC" >= "ABC" True
<=	If the second string is greater than or equal to the first, it returns True.	>>> "ABc" <= "ABc" True

```
>>> "TED"=="ted"
False
>>> "talk"=="talk"
True
>>> "talk">"talk"
False
>>> "Talk"<"talk"
True
>>>
```

PYTHON PROGRAMMING

create a list

In Python programming, a list is created by placing all the items (elements) inside a square bracket [ ], separated by commas.

It can have any number of items and they may be of different types (integer, float, string etc.).

```
# empty list
my_list = []

# list of integers
my_list = [1, 2, 3]

# list with mixed datatypes
my_list = [1, "Hello", 3.4]
```

How to access elements from a list?

There are various ways in which we can access the elements of a list.

List Index

We can use the index operator [] to access an item in a list. Index starts from 0. So, a list having 5 elements will have index from 0 to 4.

Trying to access an element other that this will raise an IndexError. The index must be an integer. We can't use float or other types, this will result into TypeError.

Nested list are accessed using nested indexing.

```
my_list = ['p','r','o','b','e']
print(my_list[0])
# Output: p

print(my_list[2])
# Output: o

print(my_list[4])
# Output: e


# Nested List
n_list = ["Happy", [2,0,1,5]]

# Nested indexing
print(n_list[0][1])
# Output: a
```

PYTHON PROGRAMMING

Negative indexing

Python allows negative indexing for its sequences. The index of -1 refers to the last item, -2 to the second last item and so on.

```
my_list = ['p','r','o','b','e']
print(my_list[-1])
# Output: e

print(my_list[-5])
# Output: p
```

Slice lists in Python

```
my_list = ['p','r','o','g','r','a','m','i','z']
# elements 3rd to 5th
print(my_list[2:5])

# elements beginning to 4th
print(my_list[:-5])

# elements 6th to end
print(my_list[5:])

# elements beginning to end
print(my_list[:])
```

To change or add elements to a list?

List are mutable, meaning, their elements can be changed unlike string or tuple.

We can use assignment operator (=) to change an item or a range of items.

```
# mistake values
odd = [2, 4, 6, 8]

# change the 1st item
odd[0] = 1
print(odd)
# Output: [1, 4, 6, 8]


# change 2nd to 4th items
odd[1:4] = [3, 5, 7]
# Output: [1, 3,5,7]
```

Python List Methods
append() - Add an element to the end of the list
extend() - Add all elements of a list to the another list

PYTHON PROGRAMMING

insert() - Insert an item at the defined index
remove() - Removes an item from the list
pop() - Removes and returns an element at the given index
clear() - Removes all items from the list
index() - Returns the index of the first matched item
count() - Returns the count of number of items passed as an argument
sort() - Sort items in a list in ascending order
reverse() - Reverse the order of items in the list
copy() - Returns a shallow copy of the list

Python List append()

The append() method adds a single item to the existing list. It doesn't return a new list; rather it modifies the original list.

The syntax of append() method is:

list.append(item)

The append() method takes a single *item* and adds it to the end of the list.

The *item* can be numbers, strings, another list, dictionary etc.

Return Value from append()

As mentioned, the append() method only modifies the original list. It doesn't return any value.

```
# animal list
animal = ['cat', 'dog', 'rabbit']

# an element is added
animal.append('guinea pig')

#Updated Animal List
print('Updated animal list: ', animal)
```

When you run the program, the output will be:

Updated animal list: ['cat', 'dog', 'rabbit', 'guinea pig']

PYTHON PROGRAMMING

Adding List to a List

```
animal = ['cat', 'dog', 'rabbit']

# another list of wild animals
wild_animal = ['tiger', 'fox']

# adding wild_animal list to animal list
animal.append(wild_animal)

#Updated List
print('Updated animal list: ', animal)
```

When you run the program, the output will be:

Updated animal list: ['cat', 'dog', 'rabbit', ['tiger', 'fox']]

Python List extend()

The extend() extends the list by adding all items of a list (passed as an argument) to the end.

The syntax of extend() method is:

```
list1.extend(list2)
```

Here, the elements of list2 are added to the end of list1.

extend() Parameters

As mentioned, the extend() method takes a single argument (a list) and adds it to the end.

If you need to add elements of other native datatypes (like tuple and set) to the list, you can simply use:

```
# add elements of a tuple to list

list.extend(list(tuple_type))
```

or even easier

```
list.extend(tuple_type)
```

Return Value from extend()

The extend() method only modifies the original list. It doesn't return any value.

```
# language list
language = ['French', 'English', 'German']

# another list of language
```

PYTHON PROGRAMMING

```
language1 = ['Spanish', 'Portuguese']

language.extend(language1)

# Extended List
print('Language List: ', language)
```
Language List: ['French', 'English', 'German', 'Spanish', 'Portuguese']

Python List insert()

The insert() method inserts the element to the list at the given index.
The syntax of insert() method is
list.insert(index, element)

insert() Parameters
The insert() function takes two parameters:
index - position where element needs to be inserted
element - this is the element to be inserted in the list

Return Value from insert()
The insert() method only inserts the element to the list. It doesn't return any value.

Python List index()

The index() method searches an element in the list and returns its index.
In simple terms, index() method finds the given element in a list and returns its position.
However, if the same element is present more than once, index() method returns its smallest/first position.

Note: Index in Python starts from 0 not 1.

The syntax of index() method for list is:
list.index(element)

index() Parameters
The index method takes a single argument:
element - element that is to be searched.

Return value from index()
The index() method returns the index of the element in the list.
If not found, it raises a ValueError exception indicating the element is not in the list.

The remove() method searches for the given element in the list and removes the first matching element.

The syntax of remove() method is:
list.remove(element)

remove() Parameters
The remove() method takes a single element as an argument and removes it from the list.
If the element(argument) passed to the remove() method doesn't exist, valueError exception is thrown.

PYTHON PROGRAMMING

Return Value from remove()
The remove() method only removes the given element from the list. It doesn't return any value.

```
animal = ['cat', 'dog', 'rabbit', 'guinea pig']

# 'rabbit' element is removed
animal.remove('rabbit')

#Updated Animal List
print('Updated animal list: ', animal)
```

When you run the program, the output will be:

Updated animal list: ['cat', 'dog', 'guinea pig']

Iterating Through a List

Using a for loop we can iterate though each item in a list.

```
for fruit in ['apple','banana','mango']:
    print("I like",fruit)
```

Output

I like apple
I like banana
I like mango

List Comprehension: Elegant way to create new List

A list comprehension is an expression and a loop with an optional condition enclosed in brackets where the loop is used to generate items for the list, and where the condition can filter out unwanted items.

The simplest form of a list comprehension is this:

```
[item for item in iterable]
```

This will return a list of every item in the iterable, and is semantically no different from list(iterable).Two things that make list comprehensions more interesting and powerful are that we can use expressions, and we can attach a condition—this takes us to the two general syntaxes for list comprehensions:

```
[expression for item in iterable] [expression for item in iterable if condition]
```

Suppose, for example, that we wanted to produce a list of the leap years in a given range. We might start out like this:

```
leaps = [] for year in range(1900, 1940):

if (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0):
leaps.append(year)
```

PYTHON PROGRAMMING

When the built-in range() function is given two integer arguments, range() ➤ 141 n and m, theiterator it returnsproducestheintegersn,n+1,…,m-1

```
pow2 = [2 ** x for x in range(10)]
print(pow2)
# Output: [1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```

```
[(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Nested List Comprehensions

The initial expression in a list comprehension can be any arbitrary expression, including another list comprehension.

Consider the following example of a 3x4 matrix implemented as a list of 3 lists of length 4:

```
>>>
>>> matrix = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12],
... ]
```

The following list comprehension will transpose rows and columns:

```
>>>
>>> [[row[i] for row in matrix] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

As we saw in the previous section, the nested listcomp is evaluated in the context of the for that follows it, so this example is equivalent to:

```
>>>
>>> transposed = []
>>> for i in range(4):
...     transposed.append([row[i] for row in matrix])
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

which, in turn, is the same as:

```
>>>
>>> transposed = []
>>> for i in range(4):
...     # the following 3 lines implement the nested listcomp
...     transposed_row = []
...     for row in matrix:
...         transposed_row.append(row[i])
...     transposed.append(transposed_row)
```

PYTHON PROGRAMMING

...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]

In the real world, you should prefer built-in functions to complex flow statements. The zip() function would do a great job for this use case:

>>>
>>> list(zip(*matrix))
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]

The del statement

There is a way to remove an item from a list given its index instead of its value: the del statement. This differs from the pop() method which returns a value. The del statement can also be used to remove slices from a list or clear the entire list (which we did earlier by assignment of an empty list to the slice). For example:

>>>
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]

del can also be used to delete entire variables:

>>>
>>> del a

Referencing the name a hereafter is an error (at least until another value is assigned to it). We'll find other uses for del later.

PYTHON PROGRAMMING

Function	Description
all()	Return True if all elements of the list are true (or if the list is empty).
any()	Return True if any element of the list is true. If the list is empty, return False.
enumerate()	Return an enumerate object. It contains the index and value of all the items of list as a tuple.
len()	Return the length (the number of items) in the list.
list()	Convert an iterable (tuple, string, set, dictionary) to a list.
max()	Return the largest item in the list.

min()	Return the smallest item in the list
sorted()	Return a new sorted list (does not sort the list itself).
sum()	Return the sum of all elements in the list.

Python enumerate()

The enumerate() method adds counter to an iterable and returns it (the enumerate object).
The syntax of enumerate() is:
enumerate(iterable, start=0)

enumerate() Parameters
The enumerate() method takes two parameters:

- iterable - a sequence, an iterator, or objects that supports iteration
- start (optional) - enumerate() starts counting from this number. If start is omitted, 0 is taken as start.

Return Value from enumerate()
The enumerate() method adds counter to an iterable and returns it. The returned object is a enumerate object.
You can convert enumerate objects to list and tuple using list() and tuple() method respectively.

Example 1: How enumerate() works in Python?

```
grocery = ['bread', 'milk', 'butter']
enumerateGrocery = enumerate(grocery)

print(type(enumerateGrocery))

# converting to list
print(list(enumerateGrocery))

# changing the default counter
enumerateGrocery = enumerate(grocery, 10)
print(list(enumerateGrocery))
```

When you run the program, the output will be:

```
<class 'enumerate'>

[(0, 'bread'), (1, 'milk'), (2, 'butter')]
[(10, 'bread'), (11, 'milk'), (12, 'butter')]
```

example 2: Looping Over an Enumerate object

```
grocery = ['bread', 'milk', 'butter']

for item in enumerate(grocery):
  print(item)

print('\n')
for count, item in enumerate(grocery):
```

PYTHON PROGRAMMING

```
  print(count, item)

print('\n')
# changing default start value
for count, item in enumerate(grocery, 100):
  print(count, item)
```

When you run the program, the output will be:

```
(0, 'bread')
(1, 'milk')
(2, 'butter')

0 bread
1 milk
2 butter

100 bread
101 milk
102 butter
```

Using Lists as Stacks

The list methods make it very easy to use a list as a stack, where the last element added is the first element retrieved (“last-in, first-out”). To add an item to the top of the stack, use append(). To retrieve an item from the top of the stack, use pop() without an explicit index. For example:

```
>>>
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

PYTHON PROGRAMMING

Using Lists as Queues

It is also possible to use a list as a queue, where the first element added is the first element retrieved ("first-in, first-out"); however, lists are not efficient for this purpose. While appends and pops from the end of list are fast, doing inserts or pops from the beginning of a list is slow (because all of the other elements have to be shifted by one).

To implement a queue, use collections.deque which was designed to have fast appends and pops from both ends. For example:

```
>>>
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")           # Terry arrives
>>> queue.append("Graham")          # Graham arrives
>>> queue.popleft()                 # The first to arrive now leaves
'Eric'
>>> queue.popleft()                 # The second to arrive now leaves
'John'
>>> queue                           # Remaining queue in order of arrival
deque(['Michael', 'Terry', 'Graham'])
```

Source code: Matrix Addition using Nested Loop

```
X = [[12,7,3],
    [4 ,5,6],
    [7 ,8,9]]

Y = [[5,8,1],
    [6,7,3],
    [4,5,9]]

result = [[0,0,0],
          [0,0,0],
          [0,0,0]]

# iterate through rows
for i in range(len(X)):
   # iterate through columns
   for j in range(len(X[0])):
       result[i][j] = X[i][j] + Y[i][j]

for r in result:
   print(r)
```

PYTHON PROGRAMMING

Source Code: Matrix Addition using Nested List Comprehension

```
# Program to add two matrices
# using list comprehension

X = [[12,7,3],
    [4 ,5,6],
    [7 ,8,9]]

Y = [[5,8,1],
    [6,7,3],
    [4,5,9]]

result = [[X[i][j] + Y[i][j]  for j in range(len(X[0]))] for i in range(len(X))]

for r in result:
   print(r)
```

Output

```
[17, 15, 4]
[10, 12, 9]
[11, 13, 18]
```

List unpacking

Although we can use the slice operator to access items in a list, in some situations we want to take two or more pieces of a list in one go. This can be done by sequence unpacking. Any iterable (lists, tuples, etc.) can be unpacked using the sequence unpacking operator, an asterisk or star(*).When used with two or more variables on the left-hand side of an assignment, one of which is preceded by * , items are assigned to the variables, with all those left over assigned to the starred variable. Here are some examples:
>>> first, *rest = [9, 2, -4, 8, 7] >>> first, rest (9, [2, -4, 8, 7])

Advantages of Tuple over List

Since, tuples are quite similiar to lists, both of them are used in similar situations as well.

However, there are certain advantages of implementing a tuple over a list. Below listed are some of the main advantages:

- We generally use tuple for heterogeneous (different) datatypes and list for homogeneous (similar) datatypes.
- Since tuple are immutable, iterating through tuple is faster than with list. So there is a slight performance boost.
- Tuples that contain immutable elements can be used as key for a dictionary. With list, this is not possible.

PYTHON PROGRAMMING

- If you have data that doesn't change, implementing it as tuple will guarantee that it remains write-protected.

  A tuple is a sequence of immutable Python objects. Tuples are sequences, just like lists. The main difference between the tuples and the lists is that the tuples cannot be changed unlike lists. Tuples use parentheses, whereas lists use square brackets.

Creating a Tuple

A tuple is created by placing all the items (elements) inside a parentheses (), separated by comma. The parentheses are optional but is a good practice to write it.

A tuple can have any number of items and they may be of different types (integer, float, list, string etc.).

```
# empty tuple
my_tuple = ()
print(my_tuple)
# Output: ()

# tuple having integers
my_tuple = (1, 2, 3)
print(my_tuple)
# Output: (1, 2, 3)


# tuple with mixed datatypes
my_tuple = (1, "Hello", 3.4)
print(my_tuple)
# Output: (1, "Hello", 3.4)


# nested tuple
my_tuple = ("mouse", [8, 4, 6], (1, 2, 3))
print(my_tuple)
# Output: ("mouse", [8, 4, 6], (1, 2, 3))


# tuple can be created without parentheses
# also called tuple packing
my_tuple = 3, 4.6, "dog"
print(my_tuple)
# Output: 3, 4.6, "dog"


# tuple unpacking is also possible
a, b, c = my_tuple
print(a)
```

PYTHON PROGRAMMING

```
print(b)
print(c)
# Output:
# 3
# 4.6
# dog
```

Creating a tuple with one element is a bit tricky.

Having one element within parentheses is not enough. We will need a trailing comma to indicate that it is in fact a tuple.

```
# only parentheses is not enough
my_tuple = ("hello")
print(type(my_tuple))
# Output: <class 'str'>


# need a comma at the end
my_tuple = ("hello",)
print(type(my_tuple))
# Output: <class 'tuple'>

# parentheses is optional
my_tuple = "hello",
print(type(my_tuple))
# Output: <class 'tuple'>
```

Accessing Elements in a Tuple

There are various ways in which we can access the elements of a tuple.

1. Indexing

We can use the index operator [] to access an item in a tuple where the index starts from 0.

So, a tuple having 6 elements will have index from 0 to 5. Trying to access an element other that (6, 7,...) will raise an IndexError.

The index must be an integer, so we cannot use float or other types. This will result into TypeError.

Likewise, nested tuple are accessed using nested indexing, as shown in the example below.

```
n_tuple = ("mouse", [8, 4, 6], (1, 2, 3))

# nested index
print(n_tuple[0][3])
# Output: 's'
```

PYTHON PROGRAMMING

Negative Indexing

Python allows negative indexing for its sequences.

The index of -1 refers to the last item, -2 to the second last item and so on.

```
my_tuple = ('p','e','r','m','i','t')
print(my_tuple[-1])
# Output: 't'


print(my_tuple[-6])
# Output: 'p'
```

Slicing

We can access a range of items in a tuple by using the slicing operator - colon ":".

```
my_tuple = ('p','r','o','g','r','a','m','i','z')

# elements 2nd to 4th
print(my_tuple[1:4])
# Output: ('r', 'o', 'g')


# elements beginning to 2nd
# Output: ('p', 'r')
print(my_tuple[:-7])

# elements 8th to end
print(my_tuple[7:])
# Output: ('i', 'z')
```

Changing a Tuple

Unlike lists, tuples are immutable.

This means that elements of a tuple cannot be changed once it has been assigned. But, if the element is itself a mutable datatype like list, its nested items can be changed.

We can also assign a tuple to different values (reassignment)

```
my_tuple = (4, 2, 3, [6, 5])
my_tuple[1] = 9

# you will get an error:
# TypeError: 'tuple' object does not support item assignment
```

PYTHON PROGRAMMING

We can use + operator to combine two tuples. This is also called concatenation.

We can also repeat the elements in a tuple for a given number of times using the * operator.

Both + and * operations result into a new tuple.

```
# Concatenation
print((1, 2, 3) + (4, 5, 6))
# Output: (1, 2, 3, 4, 5, 6)


# Repeat
print(("Repeat",) * 3)
# Output: ('Repeat', 'Repeat', 'Repeat')
```

Deleting a Tuple

As discussed above, we cannot change the elements in a tuple. That also means we cannot delete or remove items from a tuple.

But deleting a tuple entirely is possible using the keyword del.

built-in Functions with Tuple

Built-in functions like all(), any(), enumerate(), len(), max(), min(), sorted(), tuple() etc. are commonly used with tuple to perform different tasks.

Function	Description
all()	Return True if all elements of the tuple are true (or if the tuple is empty).
any()	Return True if any element of the tuple is true. If the tuple is empty, return False.
enumerate()	Return an enumerate object. It contains the index and value of all the items of tuple as pairs.
len()	Return the length (the number of items) in the tuple.
max()	Return the largest item in the tuple.
min()	Return the smallest item in the tuple
sorted()	Take elements in the tuple and return a new sorted list (does not sort the tuple itself).
sum()	Retrun the sum of all elements in the tuple.
tuple()	Convert an iterable (list, string, set, dictionary) to a tuple.

PYTHON PROGRAMMING

Named Tuples

A named tuple behaves just like a plain tuple , and has the same performance characteristics. What it adds is the ability to refer to items in the tuple by name as well as by index position, and this allows us to create aggregates of data items. The collections module provides the namedtuple() function. This function is used to create custom tuple data types. For example:

```
>>> import collections
>>> s1=collections.namedtuple("s1","pid cid price")
>>> cus=s1(12,5647,989)
>>> print(cus)
```

Output: s1(pid=12, cid=5647, price=989)

```
>>> s1=collections.namedtuple("s1",["pid","cid","price"])
>>>  val=[12,344,46545]
>>> tuple1=s1._make(val)
>>> tuple1
```

Output : s1(pid=12, cid=344, price=46545)

Set

A set type is a collection data type that supports the membership operator(in), the size function(len()),and is iterable. In addition, set type sat least provide a set. isdisjoint() method, and support for comparisons, as well as support for the bitwise operators (which in the context of sets are used for union, intersection,etc.).Python provides two built-in set types: the mutable set type and the immutable frozenset. When iterated, set types provide their items in an arbitrary order.

To create a set

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

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

```
my_set = {1,2,3,4,3,2}
print(my_set)
# Output: {1, 2, 3, 4}
```

PYTHON PROGRAMMING

```
# set cannot have mutable items
# here [3, 4] is a mutable list
# If you uncomment line #12,
# this will cause an error.
# TypeError: unhashable type: 'list'

#my_set = {1, 2, [3, 4]}

# we can make set from a list
my_set = set([1,2,3,2])
print(my_set)
# Output: {1, 2, 3}
```

Creating an empty set is a bit tricky.

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

```
# initialize a with {}
a = {}

# check data type of a
print(type(a))
# Output: <class 'dict'>

# initialize a with set()
a = set()

# check data type of a
```

PYTHON PROGRAMMING

```
print(type(a))

# Output: <class 'set'>
```

To change a set in Python

Sets are mutable. But since they are unordered, indexing have no meaning.

We cannot access or change an element of set using indexing or slicing. Set does not support it.

We can add single element using the add() method and multiple elements using the update() method. The update() method can take tuples, lists, strings or other sets as its argument. In all cases, duplicates are avoided.

```
# initialize my_set

my_set = {1,3}

print(my_set)


# if you uncomment line 9,

# you will get an error

# TypeError: 'set' object does not support indexing


#my_set[0]


# add an element

my_set.add(2)

print(my_set)

# Output: {1, 2, 3}


# add multiple elements

my_set.update([2,3,4])

print(my_set)

# Output: {1, 2, 3, 4}
```

PYTHON PROGRAMMING

```
# add list and set

my_set.update([4,5], {1,6,8})

print(my_set)

# Output: {1, 2, 3, 4, 5, 6, 8}
```

```
{1, 3}
{1, 2, 3}
{1, 2, 3, 4}
{1, 2, 3, 4, 5, 6, 8}
```

To remove elements from a set

A particular item can be removed from set using methods, discard() and remove().

The only difference between the two is that, while using discard() if the item does not exist in the set, it remains unchanged. But remove() will raise an error in such condition.

The following example will illustrate this.

```
initialize my_set

my_set = {1, 3, 4, 5, 6}

print(my_set)


# discard an element

my_set.discard(4)

print(my_set)

# Output: {1, 3, 5, 6}


# remove an element

my_set.remove(6)

print(my_set)

# Output: {1, 3, 5}


# discard an element

# not present in my_set
```

PYTHON PROGRAMMING

```
my_set.discard(2)
print(my_set)
# Output: {1, 3, 5}

# remove an element
# not present in my_set
# If you uncomment line 27,
# you will get an error.

#my_set.remove(2)
# Output: KeyError: 2
```

Similarly, we can remove and return an item using the pop() method.

Set being unordered, there is no way of determining which item will be popped. It is completely arbitrary.

We can also remove all items from a set using clear().

```
# initialize my_set
my_set = set("HelloWorld")
print(my_set)
# Output: set of unique elements

# pop an element
print(my_set.pop())
# Output: random element

# pop another element
my_set.pop()
print(my_set)
```

PYTHON PROGRAMMING

```
# Output: random element


# clear my_set
my_set.clear()
print(my_set)
#Output: set()
```

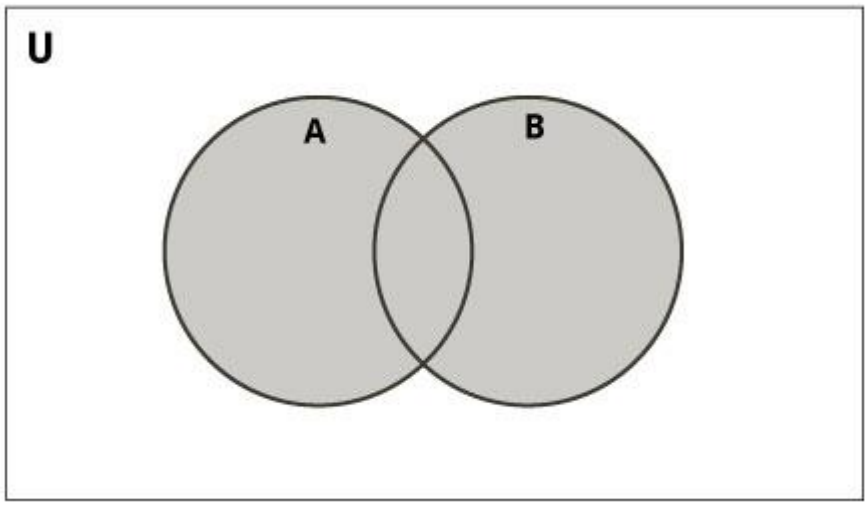
Python Set Operations

Sets can be used to carry out mathematical set operations like union, intersection, difference and symmetric difference. We can do this with operators or methods.

Let us consider the following two sets for the following operations.

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

Set Union



Union of *A* and *B* is a set of all elements from both sets.

Union is performed using | operator. Same can be accomplished using the method union().

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


# use | operator
```

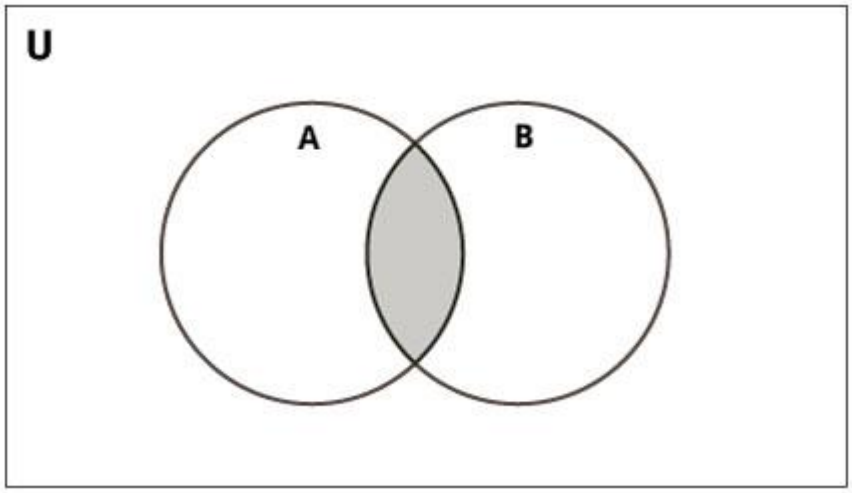
```
print(A | B)

# Output: {1, 2, 3, 4, 5, 6, 7, 8}


>> A.union(B)
{1, 2, 3, 4, 5, 6, 7, 8}

# use union function on B
>>> B.union(A)
{1, 2, 3, 4, 5, 6, 7, 8}
```

Set Intersection



Intersection of *A* and *B* is a set of elements that are common in both sets.

Intersection is performed using & operator. Same can be accomplished using the method intersection().

```
# initialize A and B

A = {1, 2, 3, 4, 5}

B = {4, 5, 6, 7, 8}


# use & operator

print(A & B)

# Output: {4, 5}
```

try the following examples on Python shell.

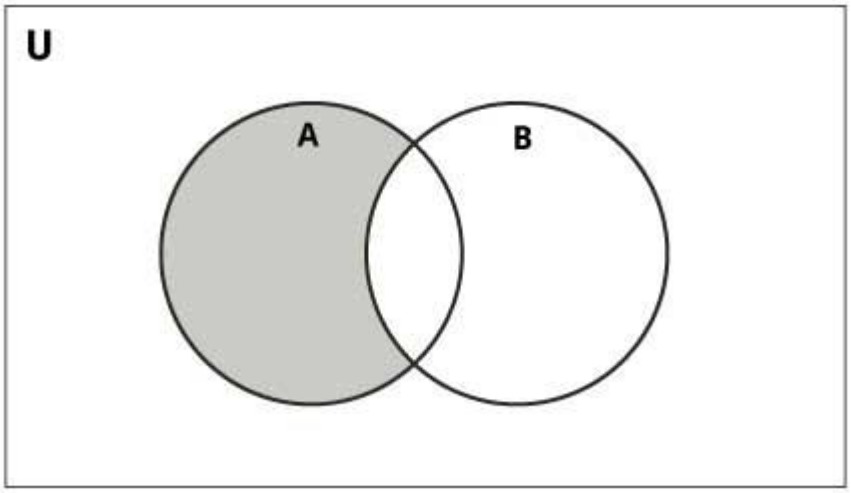
```
# use intersection function on A
```

PYTHON PROGRAMMING

```
>>> A.intersection(B)
{4, 5}

# use intersection function on B
>>> B.intersection(A)
{4, 5}
```

Set Difference



Difference of *A* and *B* (*A* - *B*) is a set of elements that are only in *A* but not in *B*. Similarly, *B* - *A* is a set of element in *B* but not in *A*.

Difference is performed using - operator. Same can be accomplished using the method difference().

```
# initialize A and B

A = {1, 2, 3, 4, 5}

B = {4, 5, 6, 7, 8}


# use - operator on A

print(A - B)

# Output: {1, 2, 3}
```

Try the following examples on Python shell.

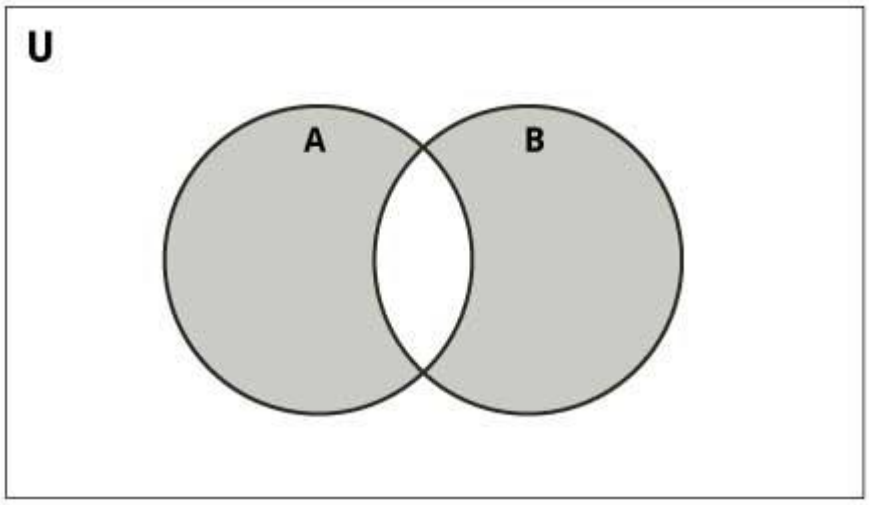
```
# use difference function on A
>>> A.difference(B)
{1, 2, 3}
```

PYTHON PROGRAMMING

```
# use - operator on B
>>> B - A
{8, 6, 7}

# use difference function on B
>>> B.difference(A)
{8, 6, 7}
```

Set Symmetric Difference



Symmetric Difference of *A* and *B* is a set of elements in both *A* and *B* except those that are common in both.

Symmetric difference is performed using ^ operator. Same can be accomplished using the method symmetric_difference().

```
# initialize A and B

A = {1, 2, 3, 4, 5}

B = {4, 5, 6, 7, 8}


# use ^ operator

print(A ^ B)

# Output: {1, 2, 3, 6, 7, 8}
```

PYTHON PROGRAMMING

Try the following examples on Python shell.

```
# use symmetric_difference function on A
>>> A.symmetric_difference(B)
{1, 2, 3, 6, 7, 8}

# use symmetric_difference function on B
>>> B.symmetric_difference(A)
{1, 2, 3, 6, 7, 8}
```

Different Python Set Methods

There are many set methods, some of which we have already used above. Here is a list of all the methods that are available with set objects.

Method	Description
add()	Add an element to a set
clear()	Remove all elements form a set
copy()	Return a shallow copy of a set
difference()	Return the difference of two or more sets as a new set
difference_update()	Remove all elements of another set from this set
discard()	Remove an element from set if it is a member. (Do nothing if the element is not in set)
intersection()	Return the intersection of two sets as a new set
intersection_update()	Update the set with the intersection of itself and another
isdisjoint()	Return True if two sets have a null intersection
issubset()	Return True if another set contains this set
issuperset()	Return True if this set contains another set
pop()	Remove and return an arbitary set element. Raise KeyError if the set is empty
remove()	Remove an element from a set. If the element is not a member, raise a KeyError
symmetric_difference()	Return the symmetric difference of two sets as a new set
symmetric_difference_update()	Update a set with the symmetric difference of itself and another
union()	Return the union of sets in a new set
update()	Update a set with the union of itself and others

PYTHON PROGRAMMING

Iterating Through a Set

Using a for loop, we can iterate though each item in a set.

```
>>> for letter in set("apple"):
...     print(letter)
```

...
a
p
e
l

Python Frozenset

Frozenset is a new class that has the characteristics of a set, but its elements cannot be changed once assigned. While tuples are immutable lists, frozensets are immutable sets.

Sets being mutable are unhashable, so they can't be used as dictionary keys. On the other hand, frozensets are hashable and can be used as keys to a dictionary.

Frozensets can be created using the function frozenset().

This datatype supports methods like copy(), difference(), intersection(), isdisjoint(), issubset(), issuperset(), symmetric_difference() and union(). Being immutable it does not have method that add or remove elements.

```
# initialize A and B

A = frozenset([1, 2, 3, 4])

B = frozenset([3, 4, 5, 6])
```

```
>>> A.isdisjoint(B)
False
>>> A.difference(B)
frozenset({1, 2})
>>> A | B
frozenset({1, 2, 3, 4, 5, 6})
>>> A.add(3)
...
AttributeError: 'frozenset' object has no attribute 'add'
```

Frozen set as dictionaries

Frozen set can be used as dictionary[hashable]

Example:

```
>>> Person={"name":"john","age": 13,"gender":"male"}

>>> fset=frozenset(Person)

>>> print(fset)
```

Output:

```
frozenset({'gender', 'age', 'name'})
```

PYTHON PROGRAMMING

Python Dictionary

Python dictionary is an unordered collection of items. While other compound data types have only value as an element, a dictionary has a key: value pair.

Dictionaries are optimized to retrieve values when the key is known.

To create a dictionary

Creating a dictionary is as simple as placing items inside curly braces {} separated by comma.

An item has a key and the corresponding value expressed as a pair, key: value.

While values can be of any data type and can repeat, keys must be of immutable type (string, number or tuple with immutable elements) and must be unique.

```
# empty dictionary
my_dict = {}

# dictionary with integer keys
my_dict = {1: 'apple', 2: 'ball'}

# dictionary with mixed keys
my_dict = {'name': 'John', 1: [2, 4, 3]}

# using dict()
my_dict = dict({1:'apple', 2:'ball'})

# from sequence having each item as a pair
my_dict = dict([(1,'apple'), (2,'ball')])
```

As you can see above, we can also create a dictionary using the built-in function dict().

To access elements from a dictionary

While indexing is used with other container types to access values, dictionary uses keys. Key can be used either inside square brackets or with the get() method.

The difference while using get() is that it returns None instead of KeyError, if the key is not found.

```
my_dict = {'name':'Jack', 'age': 26}

print(my_dict['name'])

# Output: Jack


print(my_dict.get('age'))

# Output: 26


# Trying to access keys which doesn't exist throws error
```

PYTHON PROGRAMMING

```
# my_dict.get('address')

# my_dict['address']
```

To change or add elements in a dictionary

Dictionary are mutable. We can add new items or change the value of existing items using assignment operator.

If the key is already present, value gets updated, else a new key: value pair is added to the dictionary.

```
my_dict = {'name':'Jack', 'age': 26}


# update value

my_dict['age'] = 27

print(my_dict)

#Output: {'age': 27, 'name': 'Jack'}


# add item

my_dict['address'] = 'Downtown'

print(my_dict)

# Output: {'address': 'Downtown', 'age': 27, 'name': 'Jack'}
```

When you run the program, the output will be:

```
{'name': 'Jack', 'age': 27}
{'name': 'Jack', 'age': 27, 'address': 'Downtown'}
```

To delete or remove elements from a dictionary

We can remove a particular item in a dictionary by using the method pop(). This method removes as item with the provided key and returns the value.

The method, popitem() can be used to remove and return an arbitrary item (key, value) form the dictionary. All the items can be removed at once using the clear() method.

We can also use the del keyword to remove individual items or the entire dictionary itself.

```
# create a dictionary

squares = {1:1, 2:4, 3:9, 4:16, 5:25}
```

PYTHON PROGRAMMING

```
# remove a particular item
print(squares.pop(4))
# Output: 16

print(squares)
# Output: {1: 1, 2: 4, 3: 9, 5: 25}


# remove an arbitrary item
print(squares.popitem())
# Output: (1, 1)

print(squares)
# Output: {2: 4, 3: 9, 5: 25}

# delete a particular item
del squares[5]

print(squares)
# Output: {2: 4, 3: 9}

# remove all items
squares.clear()

print(squares)
# Output: {}
```

PYTHON PROGRAMMING

```
# delete the dictionary itself
del squares


# Throws Error
# print(squares)
```

When you run the program, the output will be:

```
16
{1: 1, 2: 4, 3: 9, 5: 25}
(1, 1)
{2: 4, 3: 9, 5: 25}
{2: 4, 3: 9}
{}
```

Python Dictionary Methods

Methods that are available with dictionary are tabulated below. Some of them have already been used in the above examples.

Method	Description
clear()	Remove all items form the dictionary.
copy()	Return a shallow copy of the dictionary.
fromkeys(*seq*[, *v*])	Return a new dictionary with keys from *seq* and value equal to *v* (defaults to None).
get(*key*[,*d*])	Return the value of *key*. If *key* doesnot exit, return *d* (defaults to None).
items()	Return a new view of the dictionary's items (key, value).
keys()	Return a new view of the dictionary's keys.
pop(*key*[,*d*])	Remove the item with *key* and return its value or *d* if *key* is not found. If *d* is not provided and *key* is not found, raises KeyError.
popitem()	Remove and return an arbitary item (key, value). Raises KeyError if the dictionary is empty.
setdefault(*key*[,*d*])	If *key* is in the dictionary, return its value. If not, insert *key* with a value of *d* and return *d* (defaults to None).
update([*other*])	Update the dictionary with the key/value pairs from *other*, overwriting existing keys.
values()	Return a new view of the dictionary's values

```
marks = {}.fromkeys(['Math','English','Science'], 0)


# Output: {'English': 0, 'Math': 0, 'Science': 0}
```

PYTHON PROGRAMMING

```
print(marks)


for item in marks.items():
    print(item)


list(sorted(marks.keys()))
# Output: ['English', 'Math', 'Science']
```

Python Dictionary Comprehension

Dictionary comprehension is an elegant and concise way to create new dictionary from an iterable in Python.

Dictionary comprehension consists of an expression pair (key: value) followed by for statement inside curly braces {}.

Here is an example to make a dictionary with each item being a pair of a number and its square.

```
squares = {x: x*x for x in range(6)}
print(squares)
# Output: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

This code is equivalent to

```
squares = {}
for x in range(6):
   squares[x] = x*x
```

A dictionary comprehension can optionally contain more for or if statements.

An optional if statement can filter out items to form the new dictionary.

Here are some examples to make dictionary with only odd items

```
odd_squares = {x: x*x for x in range(11) if x%2 == 1}
print(odd_squares)
# Output: {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}
```

PYTHON PROGRAMMING

Iterating Through a Dictionary

Using a for loop we can iterate though each key in a dictionary.

```
>>> squares = {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}
>>> for i in squares:
        print(squares[i])
```

```
1
9
81
25
49
```

```
# To print only key in dictionaries
for i in squares:
    print(k)
```

```
1
3
5
7
9
```

```
#To print only values in dictionaries
for v in squares.values():
    print(v)
```

```
1
9
```

PYTHON PROGRAMMING

```
81
25
49
#To print both key,values in dictionaries
for k,v in squares.items():
        print(k,v)



1 1
3 9
9 81
5 25
7 49
```

Nested dictionaries:

```
>>> students={"shiva":{"cs":90,"ds":100},"aravinth":{ "cs":100,"ds":90}}
>>> for k,v in students.items():
        print(k,v)



shiva {'ds': 100, 'cs': 90}
aravinth {'ds': 90, 'cs': 100}
```

PYTHON PROGRAMMING