


Python Material

1.	Python Introduction
2.	What is Python
3.	Python Features
4.	Python History
5.	Python Version
6.	Python Applications
7.	Python Install
8.	Python Path
9.	Python ExampleExecute Python
10.	Python Variables
11.	Python Keywords
12.	Python Identifiers
13.	Python Literals
14.	Python Operators and Python Comments
15.	Control Statement
16.	Python IfPython If elsePython else ifPython nested ifPython for loop
17.	Python while loop
18.	Python break
19.	Python continue
20.	Python pass 
21.	<u>Python Strings</u>
22.	<u>Python Lists</u>
23.	<u>Python Tuples</u>
24.	<u>Python Dictionary</u>
25.	<u>Python Functions</u>

26.	<u>Python Files I/O</u>
27.	<u>Python Modules</u>
28.	<u>Python Exceptions</u>
29.	<u>Python Date</u>
30.	Python Programs

Python Introduction

Python is a general purpose, dynamic, high level and interpreted programming language. It supports Object Oriented programming approach to develop applications. It is simple and easy to learn and provides lots of high-level data structures.

Python is *easy to learn* yet powerful and versatile scripting language which makes it attractive for Application Development.

Python's syntax and *dynamic typing* with its interpreted nature, makes it an ideal language for scripting and rapid application development.

Python supports *multiple programming pattern*, including object oriented, imperative and functional or procedural programming styles.

Python is not intended to work on special area such as web programming. That is why it is known as *multipurpose* because it can be used with web, enterprise, 3D CAD etc.

We don't need to use data types to declare variable because it is *dynamically typed* so we can write `a=10` to assign an integer value in an integer variable.

Python makes the development and *debugging fast* because there is no compilation step included in python development and edit-test-debug cycle is very fast.

Python History

- Python laid its foundation in the late 1980s.
- The implementation of Python was started in the December 1989 by **Guido Van Rossum** at CWI in Netherland.
- In February 1991, van Rossum published the code (labeled version 0.9.0) to alt.sources.

- In 1994, Python 1.0 was released with new features like: lambda, map, filter, and reduce.
- Python 2.0 added new features like: list comprehensions, garbage collection system.
- On December 3, 2008, Python 3.0 (also called "Py3K") was released. It was designed to rectify fundamental flaw of the language.
- *ABC programming language* is said to be the predecessor of Python language which was capable of Exception Handling and interfacing with Amoeba Operating System.
- Python is influenced by following programming languages:
 - ABC language.
 - Modula-3

Python Version

Python programming language is being updated regularly with new features and supports. There are lots of updates in python versions, started from 1994 to current release.

Python Version	Released Date
Python 1.0	January 1994
Python 1.5	December 31, 1997
Python 1.6	September 5, 2000
Python 2.0	October 16, 2000
Python 2.1	April 17, 2001

Python 2.2	December 21, 2001
Python 2.3	July 29, 2003
Python 2.4	November 30, 2004
Python 2.5	September 19, 2006
Python 2.6	October 1, 2008
Python 2.7	July 3, 2010
Python 3.0	December 3, 2008
Python 3.1	June 27, 2009`
Python 3.2	February 20, 2011
Python 3.3	September 29, 2012
Python 3.4	March 16, 2014
Python 3.5	September 13, 2015
Python 3.6	December 23, 2016
Python 3.6.4	December 19, 2017

Python Applications Area

Python is known for its general-purpose nature that makes it applicable in almost each domain of software development. Python as a whole can be used in any sphere of development.

Here, we are specifying applications areas where python can be applied.

1) **Web Applications**

We can use Python to develop web applications. It provides libraries to handle internet protocols such as HTML and XML, JSON, Email processing, request, beautiful Soup, Feed parser etc. It also provides Frameworks such as Django, Pyramid, Flask etc to design and develop web based applications. Some important developments are: Python Wiki Engines, Pocoo, Python Blog Software etc.

2) **Desktop GUI Applications**

Python provides Tk GUI library to develop user interface in python based application. Some other useful toolkits wxWidgets, Kivy, pyqt that are useable on several platforms. The Kivy is popular for writing multitouch applications.

3) **Software Development**

Python is helpful for software development process. It works as a support language and can be used for build control and management, testing etc.

4) **Scientific and Numeric**

Python is popular and widely used in scientific and numeric computing. Some useful library and package are SciPy, Pandas, IPython etc. SciPy is group of packages of engineering, science and mathematics.

5) **Business Applications**

Python is used to build Business applications like ERP and e-commerce systems. Tryton is a high level application platform.

6) **Console Based Application**

We can use Python to develop console based applications. For example: **IPython**.

7) Audio or Video based Applications

Python is awesome to perform multiple tasks and can be used to develop multimedia applications. Some of real applications are: TimPlayer, cplay etc.

8) 3D CAD Applications

To create CAD application Fandango is a real application which provides full features of CAD.

9) Enterprise Applications

Python can be used to create applications which can be used within an Enterprise or an Organization. Some real time applications are: OpenErp, Tryton, Picalo etc.

10) Applications for Images

Using Python several application can be developed for image. Applications developed are: VPython, Gogh, imgSeek etc.

There are several such applications which can be developed using Python

SETTING PATH IN PYTHON

Before starting working with Python, a specific path is to set.

- Your Python program and executable code can reside in any directory of your system, therefore Operating System provides a specific search path that index the directories Operating System should search for executable code.
- The Path is set in the Environment Variable of My Computer properties:
- To set path follow the steps:

Right click on My Computer ->Properties ->Advanced System setting ->Environment Variable ->New

In Variable name write path and in Variable value copy path up to C://Python(i.e., path where Python is installed). Click Ok ->Ok.

Path will be set for executing Python programs.

1. Right click on My Computer and click on properties.

2. Click on Advanced System settings

Python Example

Python is easy to learn and code and can be execute with python interpreter. We can also use Python interactive shell to test python code immediately.

A simple hello world example is given below. Write below code in a file and save with **.py** extension. Python source file has **.py** extension.

```
hello.py
```

```
print("hello world by python!")
```

Execute this example by using following command.

```
Python3 hello.py
```

After executing, it produces the following output to the screen.

Output

```
hello world by python!
```


Python Example using Interactive Shell:

Python interactive shell is used to test the code immediately and does not require to write and save code in file.

Python code is simple and easy to run. Here is a simple Python code that will print "Welcome to Python".

A simple python example is given below.

```
>>> a="Welcome To Python"
>>> print a
Welcome To Python
>>>
```

How to execute python

To execute Python code, we can use any approach that are given below.

1) Interactive Mode

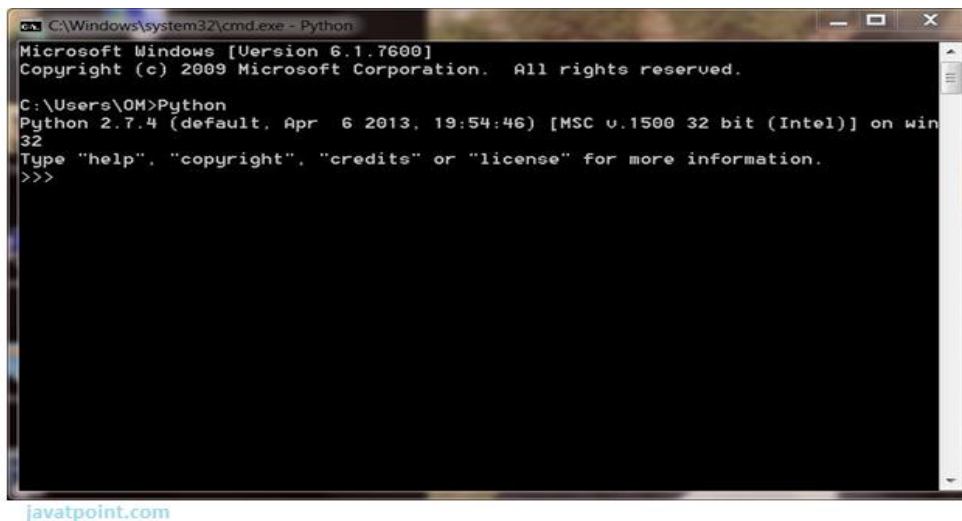
Python provides Interactive Shell to execute code immediately and produce output instantly. To get into this shell, write python in the command prompt and start working with Python.



javatpoint.cpm

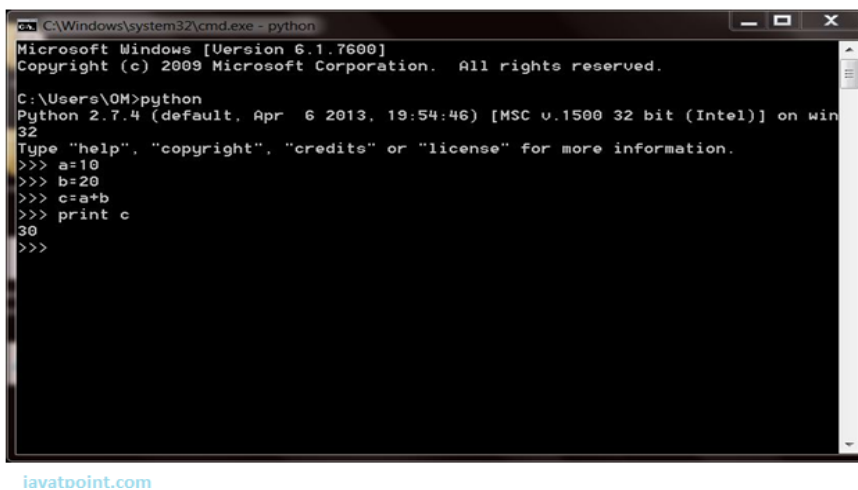
write python code:

Press Enter key and the Command Prompt will appear like:



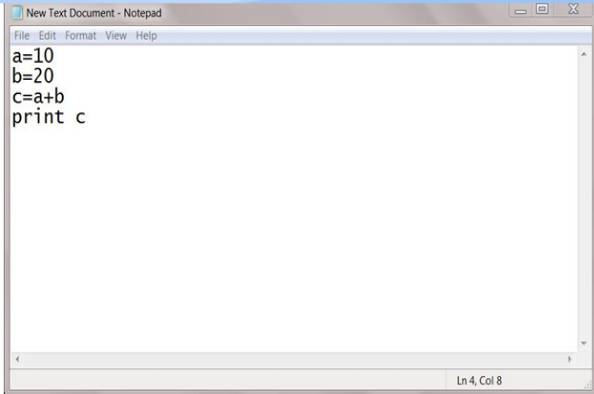
Now we can execute our Python commands.

Example:



2) Script Mode

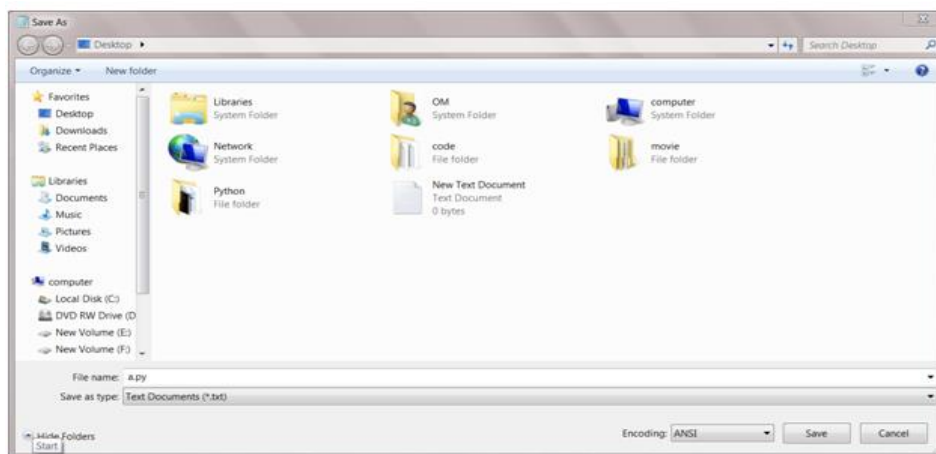
Using Script Mode, we can write our Python code in a separate file of any editor in our Operating System.



```
a=10
b=20
c=a+b
print c
```

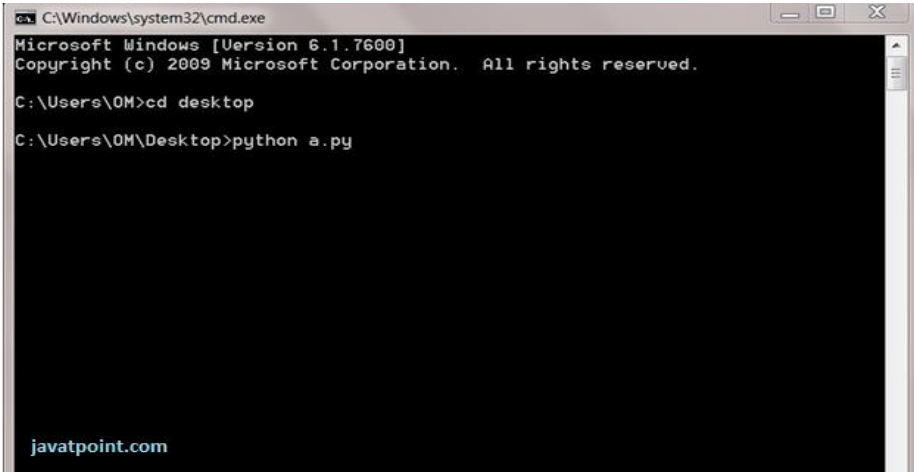
javatpoint.com

Save it by .py extension.



javatpoint.com

Now open Command prompt and execute it by :



```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\OM>cd desktop
C:\Users\OM\Desktop>python a.py
```

javatpoint.com

Python Variables:

Variable is a name which is used to refer memory location. Variable also known as identifier and used to hold value.

In Python, we don't need to specify the type of variable because Python is a type infer language and smart enough to get variable type.

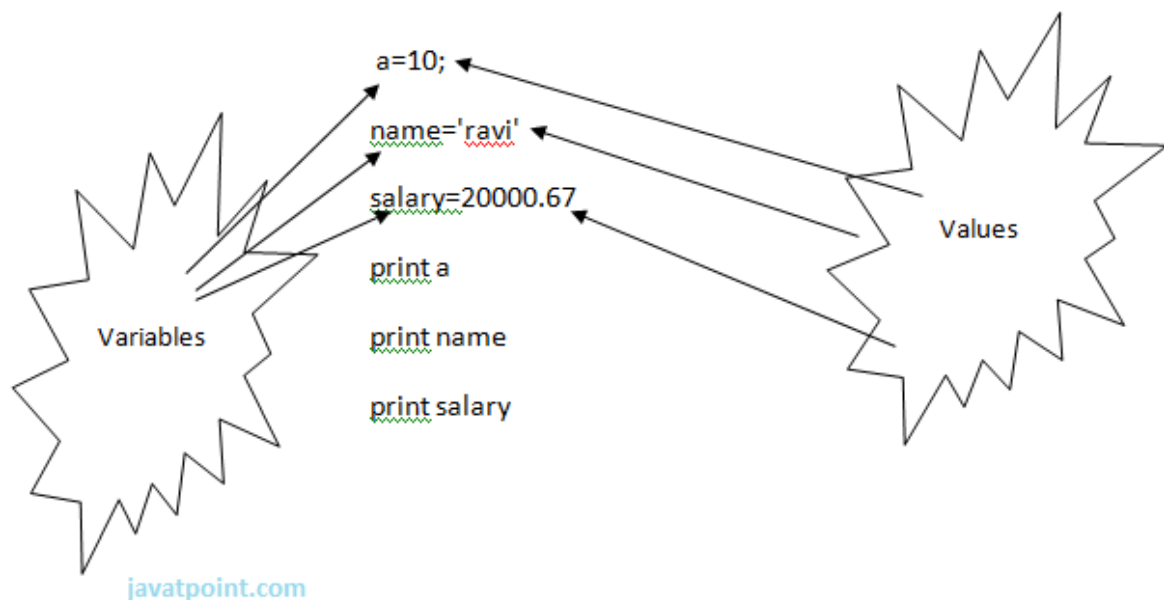
Variable names can be a group of both letters and digits, but they have to begin with a letter or an underscore.

Declaring Variable and Assigning Values:

Python does not bound us to declare variable before using in the application. It allows us to create variable at required time.

We don't need to declare explicitly variable in Python. When we assign any value to the variable that variable is declared automatically.

The equal (=) operator is used to assign value to a variable.



Output:

```
>>>
10
ravi
20000.67
>>>
```

Multiple Assignment:

Python allows us to assign a value to multiple variables in a single statement which is also known as multiple assignment.

We can apply multiple assignments in two ways either by assigning a single value to multiple variables or assigning multiple values to multiple variables.

1.Assigning single value to multiple variables

```
x=y=z=50
print x
print y
print z
```

Output:

```
>>>
50
50
50
>>>
```

2.Assigning multiple values to multiple variables:

```
a,b,c=5,10,15

print a
```

```
print b  
print c
```

Output:

```
>>>  
5  
10  
15  
>>>
```

Basic Fundamentals:

This section contains the basic fundamentals of Python like :

i) Tokens and their types.**ii) Comments****a) Tokens:**

- Tokens can be defined as a punctuator mark, reserved words and each individual word in a statement.
- Token is the smallest unit inside the given program.

There are following tokens in Python:

- Keywords.
- Identifiers.
- Literals.
- Operators.

Tuples:

- Tuple is another form of collection where different type of data can be stored.

- It is similar to list where data is separated by commas. Only the difference is that list uses square bracket and tuple uses parenthesis.
- Tuples are enclosed in parenthesis and cannot be changed.

Example:

```
>>> tuple=('rahul',100,60.4,'deepak')
>>> tuple1=('sanjay',10)
>>> tuple
('rahul', 100, 60.4, 'deepak')
>>> tuple[2:]
(60.4, 'deepak')
>>> tuple1[0]
'sanjay'
>>> tuple+tuple1
('rahul', 100, 60.4, 'deepak', 'sanjay', 10)
>>>
```

Dictionary:

- Dictionary is a collection which works on a key-value pair.
- It works like an associated array where no two keys can be same.
- Dictionaries are enclosed by curly braces ({}) and values can be retrieved by square bracket([]).

Example:

```
>>> dictionary={'name':'charlie','id':100,'dept':'it'}
>>> dictionary
{'dept': 'it', 'name': 'charlie', 'id': 100}
>>> dictionary.keys()
['dept', 'name', 'id']
>>> dictionary.values()
```

```
['it', 'charlie', 100]
```

```
>>>
```

Python Keywords

Python Keywords are special reserved words which convey a special meaning to the compiler/interpreter. Each keyword have a special meaning and a specific operation. These keywords can't be used as variable. Following is the List of Python Keywords.

List of Python Keywords:

True	False	None	and	as
asset	Def	class	continue	break
else	Finally	elif	del	except
global	For	if	from	import
raise	Try	or	return	pass
nonlocal	In	not	is	lambda

Identifiers

Identifiers are the names given to the fundamental building blocks in a program.

These can be variables, class, object, functions, lists, dictionaries etc.

There are certain rules defined for naming i.e., Identifiers.

An identifier is a long sequence of characters and numbers.

No special character except underscore (_) can be used as an identifier.

Keyword should not be used as an identifier name.

Python is case sensitive. So, using case is significant.

First character of an identifier can be character, underscore (_) but not digit.

Python Literals

Literals can be defined as a data that is given in a variable or constant.

Python support the following literals:

I. String literals:

String literals can be formed by enclosing a text in the quotes. We can use both single as well as double quotes for a String.

Example:

```
"Aman", '12345'
```

Types of Strings:

There are two types of Strings supported in Python:

a). Single line String- Strings that are terminated within a single line are known as Single line Strings.

Example:

```
1. >>> text1='hello'
```

b). Multi line String- A piece of text that is spread along multiple lines is known as Multiple line String.

There are two ways to create Multiline Strings:

1). Adding black slash at the end of each line.

Example:

```
>>> text1='hello\  
user'  
>>> text1  
'hellouser'  
>>>
```

2). Using triple quotation marks: -

Example:

```
1. >>> str2="""welcome
2. to
3. SSSIT"""
4. >>> print str2
5. welcome
6. to
7. SSSIT
8. >>>
```

List:

- List contain items of different data types. Lists are mutable i.e., modifiable.
- The values stored in List are separated by commas (,) and enclosed within a square bracket ([]). We can store different type of data in a List.
- Value stored in a List can be retrieved using the slice operator ([] and [:]).
- The plus sign (+) is the list concatenation and asterisk (*) is the repetition operator.

Eg:

```
>>> list=['aman',678,20.4,'saurav']
>>> list1=[456,'rahul']
>>> list
['aman', 678, 20.4, 'saurav']
>>> list[1:3]
[678, 20.4]
>>> list+list1
['aman', 678, 20.4, 'saurav', 456, 'rahul']
>>> list1*2
[456, 'rahul', 456, 'rahul']
```

```
>>>
```

Python Operators

Operators are particular symbols that are used to perform operations on operands. It returns result that can be used in application.

Example

$4 + 5 = 9$

Here 4 and 5 are Operands and (+), (=) signs are the operators. This expression produces the output 9.

Types of Operators

Python supports the following operators

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

8. Arithmetic Operators

The following table contains the arithmetic operators that are used to perform arithmetic operations.

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

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

Example

1. >>> 10+20
2. 30
3. >>> 20-10
4. 10
5. >>> 10*2
6. 20
7. >>> 10/2
8. 5
9. >>> 10%3
10. 1
11. >>> 2**3
12. 8
13. >>> 10//3
14. 3
15. >>>

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 !=)

The following table contains the relational operators that are used to check relations.

example:

```
>>> 10<20
True
>>> 10>20
False
>>> 10<=10
True
>>> 20>=15
True
>>> 5==6
False
>>> 5!=6
True
>>> 10<>2
True
```

```
>>>
```

Assignment Operators:

The following table contains the assignment operators that are used to assign values to the variables.

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

Example

```
>>> c=10
>>> c
10
>>> c+=5
>>> c
15
>>> c/=2
>>> c
10
>>> c%=3
>>> c
```

```
1
>>> c=5
>>> c**=2
>>> c
25
>>> c//=2
>>> c
12
>>>
```

Logical Operators

The following table contains the arithmetic operators that are used to perform arithmetic operations.

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)

Example

1. a=5>4 **and** 3>2
2. **print** a
3. b=5>4 **or** 3<2
4. **print** b
5. c=**not**(5>4)
6. **print** c

Output:

1. >>>
2. True
3. True
4. False
5. >>>

Membership Operators:

The following table contains the 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.

Example

```
a=10
b=20
list=[10,20,30,40,50];
if (a in list):
    print "a is in given list"
else:
    print "a is not in given list"
if(b not in list):
    print "b is not given in list"
else:
    print "b is given in list"
```

Output:

```
>>>
a is in given list
b is given in list
>>>
```

Identity Operators

The following table contains the identity operators.

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
```

Output

```
>>>
a,b have same identity
a,b have different identity
>>>
```

Python If Statements:

The Python if statement is a statement which is used to test specified condition. We can use if statement to perform conditional operations in our Python application.

The if statement executes only when specified condition is **true**. We can pass any valid expression into the if parentheses.

There are various types of if statements in Python.

- if statement
- if-else statement
- nested if statement

Python If Statement Syntax

1. **if**(condition):
2. statements

Python If Statement Example

```
a=10
```

```
if a==10:
```

```
    print "Welcome to javatpoint"
```

Output:

```
Hello User
```

Python If Else Statements :

The If statement is used to test specified condition and if the condition is true, if block executes, otherwise else block executes.

The else statement executes when the if statement is false.

Python If Else Syntax

```
if(condition): False
```

```
    statements
```

```
    else: True 0
```

```
    statements
```

Write a program to find Leap year

1. year=2000
2. **if** year%4==0:
3. **print** "Year is Leap"
4. **else:**
5. **print** "Year is not Leap"

Output:

Year is Leap

Python Nested If Else Statement

In python, we can use nested If Else to check multiple conditions. Python provides **elif** keyword to make nested If statement.

This statement is like executing a if statement inside a else statement.

Python Nested If Else Syntax

1. If statement:
2. Body
3. **elif** statement:
4. Body
5. **else**:
6. Body

Python Nested If Else Example

1. a=10
2. **if** a>=20:
3. **print** "Condition is True"
4. **else**:
5. **if** a>=15:
6. **print** "Checking second value"
7. **else**:
8. **print** "All Conditions are false"

Output:

All Conditions are false.

For Loop

Python **for loop** is used to iterate the elements of a collection in the order that they appear. This collection can be a sequence (list or string).

Python for Loop Syntax

for <variable> **in** <sequence>:

Output:

- 1.
- 7
- 9

Explanation:

- Firstly, the first value will be assigned in the variable.
- Secondly all the statements in the body of the loop are executed with the same value.
- Thirdly, once step second is completed then variable is assigned the next value in the sequence and step second is repeated.
- Finally, it continues till all the values in the sequence are assigned in the variable and processed.

Python For Loop Simple Example

1. num=2
2. **for** a **in** range (1,6):
3. **print** num * a

Output: 2,4,6,8,10

Python Example to Find Sum of 10 Numbers

```
sum=0
for n in range(1,11):
    sum+=n
print sum
```

Output: 55

Python Nested For Loops

Loops defined within another Loop are called Nested Loops. Nested loops are used to iterate matrix elements or to perform complex computation.

When an outer loop contains an inner loop in its body it is called Nested Looping.

Python Nested For Loop Syntax

1. **for** <expression>:
2. **for** <expression>:
3. Body

Python Nested For Loop Example

1. **for i in** range(1,6):
2. **for j in** range (1,i+1):
3. **print** i,
4. **print**

Output:

1. >>>
2. 1
3. 2 2
4. 3 3 3

5. 4 4 4 4
6. 5 5 5 5 5
7. >>>

Explanation:

For each value of Outer loop the whole inner loop is executed.

For each value of inner loop the Body is executed each time.

Python Nested Loop Example 2

1. **for** i **in** range (1,6):
2. **for** j **in** range (5,i-1,-1):
3. **print** "*",
4. **print**

Output:

1. >>>
2. * * * * *
3. * * * * *
4. * * *
5. * *
6. *

Python While Loop

In Python, while loop is used to execute number of statements or body till the specified condition is true. Once the condition is false, the control will come out of the loop.

Python While Loop Syntax

1. **while** <expression>:
2. Body

Here, loop Body will execute till the expression passed is true. The Body may be a single statement or multiple statement.

Python While Loop Example 1

1. a=10
2. **while** a>0:
3. **print** "Value of a is",a
4. a=a-2

print "Loop is Completed"

Output:

1. >>>
2. Value of a **is** 10
3. Value of a **is** 8
4. Value of a **is** 6
5. Value of a **is** 4
6. Value of a **is** 2
7. Loop **is** Completed
8. >>>

Python While Loop Example 2

```
n=153
sum=0
while n>0:
    r=n%10
    sum+=r
    n=n/10
print sum
```

Output:

```
>>> 9 >>>
```

Python Break

Break statement is a jump statement which is used to transfer execution control. It breaks the current execution and in case of inner loop, inner loop terminates immediately.

When break statement is applied the control points to the line following the body of the loop, hence applying break statement makes the loop to terminate and controls goes to next line pointing after loop body.

Python Break Example 1

```
1. for i in [1,2,3,4,5]:  
2.     if i==4:  
3.         print "Element found"  
4.         break  
5.     print i,
```

Output:

```
1. >>>  
2. 1 2 3 Element found  
3. >>>
```

Python Break Example 2

```
1. for letter in 'Python3':  
2.     if letter == 'o':  
3.         break  
4.     print (letter)
```

Output:

```
1. P  
2. y  
3. t ,h
```


Python Continue Statement:

Python Continue Statement is a jump statement which is used to skip execution of current iteration. After skipping, loop continue with next iteration.

We can use continue statement with for as well as while loop in Python.

Python Continue Statement Example

```
a=0
while a<=5:
    a=a+1
    if a%2==0:
        continue
    print a
print "End of Loop"
```

Output:

```
>>>
1 ,3 ,5
End of Loop
>>>
```

Python Pass

In Python, pass keyword is used to execute nothing; it means, when we don't want to execute code, the pass can be used to execute empty.

Python Pass Syntax

pass

Python Pass Example

```
for i in [1,2,3,4,5]:
    if i==3:
        pass
```

```
print "Pass when value is",i  
print i,
```

Output:

```
>>>  
1 2 Pass when value is 3 4 5  
>>>
```

PYTHON STRINGS

Python string is a built-in type text sequence. It is used to handle textual data in python. Python Strings are immutable sequences of Unicode points. Creating Strings are simplest and easy to use in Python.

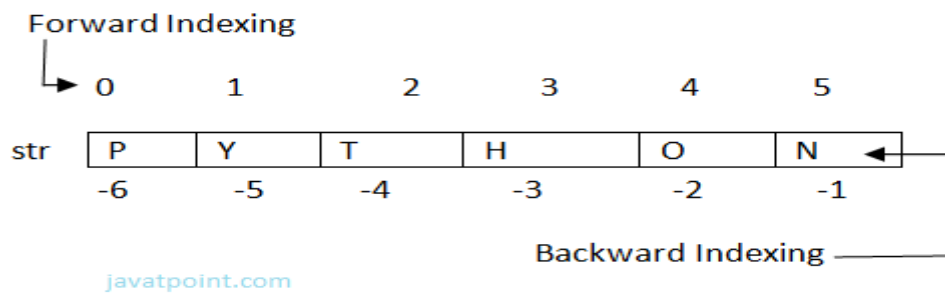
We can simply create Python String by enclosing a text in single as well as double quotes. Python treat both single and double quotes statements same.

Accessing Python Strings

In Python, Strings are stored as individual characters in a contiguous memory location.

- The benefit of using String is that it can be accessed from both the directions (forward and backward).
- Both forward as well as backward indexing are provided using Strings in Python.
 - Forward indexing starts with 0,1,2,3,....
 - Backward indexing starts with -1,-2,-3,-4,....

Example



1. `str[0]='P'=str[6]` , `str[1]='Y' = str[5]` , `str[2] = 'T' = str[4]` , `str[3] = 'H' = str[-3]` , `str[4] = 'O' = str[-2]` , `str[5] = 'N' = str[-1]`.

Python String Example:

Here, we are creating a simple program to retrieve String in reverse as well as normal form.

1. `name="Rajat"`
2. `length=len(name)`
3. `i=0`
4. **for** `n in range(-1,(-length-1),-1):`
5. **print** `name[i],"\t",name[n]`
6. `i+=1`

Output:

```
>>>
```

```
R      t
a      a
j      j
a      a
t      R
>>>
```

Python Strings Operators:

To perform operation on string, Python provides basically 3 types of Operators that are given below.

1. Basic Operators.

2. Membership Operators.
3. Relational Operators.

Python String Basic Operators

There are two types of basic operators in String "+" and "*".

String Concatenation Operator (+)

The concatenation operator (+) concatenates two Strings and creates a new String.

Python String Concatenation Example

```
>>> "ratan" + "jaiswal"
```

Output:

```
'ratanjaiswal'  
>>>
```

Expression	Output
'10' + '20'	'1020'
"s" + "007"	's007'
'abcd123' + 'xyz4'	'abcd123xyz4'

Eg: 'abc' + 3

```
>>>
```

output:

Traceback (most recent call last):

```
File "", line 1, in  
  'abc' + 3
```

TypeError: cannot concatenate 'str' and 'int' objects

```
>>>
```

Python String Replication Operator (*)

Expression	Output
"soono"*2	'soonosoono'
'1'*3	'111'
'\$'*5	'\$\$\$\$\$'

Replication operator uses two parameters for operation, One is the integer value and the other one is the String argument.

The Replication operator is used to repeat a string number of times. The string will be repeated the number of times which is given by the integer value.

Python String Replication Example:

1. >>> 5*"Vimal"

Output:

```
'VimalVimalVimalVimalVimal'
```

Python List:

List in python is implemented to store the sequence of various type of data. However, python contains six data types that are capable to store the sequences but the most common and reliable type is list.

A list can be defined as a collection of values or items of different types. The items in the list are separated with the comma (,) and enclosed with the square brackets [].

A list can be defined as follows.

```
L1 = ["John", 102, "USA"]
```

```
L2 = [1, 2, 3, 4, 5, 6]
```

```
L3 = [1, "Ryan"]
```

WAP To Define a list and printing its values.

```
emp = ["John", 102, "USA"]
Dep1 = ["CS",10];
Dep2 = ["IT",11];
HOD_CS = [10,"Mr. Holding"]
HOD_IT = [11, "Mr. Bewon"]
print("printing employee data...");
print("Name : %s, ID: %d, Country: %s"%(emp[0],emp[1],emp[2]))
print("printing departments...");
print("Department 1:\nName: %s, ID: %d\nDepartment 2:\nName: %s,
ID: %s"%(Dep1[0],Dep2[1],Dep2[0],Dep2[1]));
print("HOD Details ....");
print("CS HOD Name: %s, Id: %d"%(HOD_CS[1],HOD_CS[0]));
print("IT HOD Name: %s, Id: %d"%(HOD_IT[1],HOD_IT[0]));
print(type(emp),type(Dep1),type(Dep2),type(HOD_CS),type(HOD_IT));
```

OUTPUT

```
printing employee data...
Name : John, ID: 102, Country: USA
printing departments...
Department 1:
Name: CS, ID: 11
Department 2:
Name: IT, ID: 11
HOD Details ....
CS HOD Name: Mr. Holding, Id: 10
IT HOD Name: Mr. Bewon, Id: 11
<class 'list'> <class 'list'> <class 'list'> <class 'list'>
<class 'list'>
```

List indexing and splitting:

The indexing are processed in the same way as it happens with the strings.

The elements of the list can be accessed by using the slice operator `[]`.

The index starts from 0 and goes to length - 1. The first element of the list is stored at the 0th index, the second element of the list is stored at the 1st index, and so on.

List = [0, 1, 2, 3, 4, 5]

0	1	2	3	4	5
---	---	---	---	---	---

List[0] = 0 List[0:] = [0,1,2,3,4,5]

List[1] = 1 List[:] = [0,1,2,3,4,5]

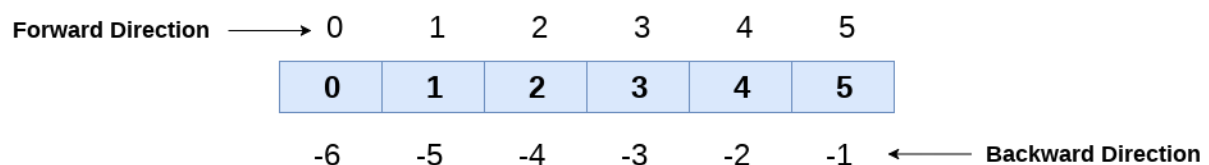
List[2] = 2 List[2:4] = [2, 3]

List[3] = 3 List[1:3] = [1, 2]

List[4] = 4 List[:4] = [0, 1, 2, 3]

List[5] = 5

List = [0, 1, 2, 3, 4, 5]



Updating List values

Lists are the most versatile data structures in python since they are immutable and their values can be updated by using the slice and assignment operator.

Python also provide us the `append()` method which can be used to add values to the string.

1. List = [1, 2, 3, 4, 5, 6]
2. **print**(List)
3. List[2] = 10;
4. **print**(List)
5. List[1:3] = [89, 78]
6. **print**(List)

Output:

```
[1, 2, 3, 4, 5, 6]
[1, 2, 10, 4, 5, 6]
[1, 89, 78, 4, 5, 6]
```

The list elements can also be deleted by using the **del** keyword. Python also provides us the **remove()** method if we do not know which element is to be deleted from the list.

Consider the following example to delete the list elements.

1. List = [0,1,2,3,4]
2. **print**(List)
3. **del** List[0]
4. **print**(List)
5. **del** List[3]
6. **print**(List)

Output:

```
[0, 1, 2, 3, 4]
[1, 2, 3, 4]
[1, 2, 3]
```

Python List Operations:

The concatenation (+) and repetition (*) operator work in the same way as they were working with the strings.

1. Consider a List l1 = [1, 2, 3, 4], **and** l2 = [5, 6, 7, 8]

Operator	Description	Example
Repetition	The repetition operator enables the list elements to be repeated multiple times.	$L1 * 2 = [1, 2, 3, 4, 1, 2, 3, 4]$
Concatenation	It concatenates the list mentioned on either side of the operator.	$l1 + l2 = [1, 2, 3, 4, 5, 6, 7, 8]$
Membership	It returns true if a particular item exists in a particular list otherwise false.	<code>print(2 in l1)</code> prints True.
Iteration	The for loop is used to iterate over the list elements.	<code>for i in l1:</code> <code>print(i)</code> Output 1,2,3,4
Length	It is used to get the length of the list	<code>len(l1) = 4</code>

Iterating a List

A list can be iterated by using a for - in loop. A simple list containing four strings can be iterated as follows.

```
List = ["John", "David", "James", "Jonathan"]
```

```
for i in List:
```

```
    print(i);
```

Output:

John

David

James

Jonathan

Adding elements to the list

Python provides `append()` function by using which we can add an element to the list. However, the `append()` method can only add the value to the end of the list.

1. `l = [];`
2. `n = int(input("Enter the number of elements in the list")); #Number of elements will be entered by the user`
3. `for i in range(0,n): # for loop to take the input`
4. `l.append(input("Enter the item?")); # The input is taken from the user and added to the list as the item`
5. `print("printing the list items....");`
6. `for i in l: # traversal loop to print the list items`
7. `print(i, end = " ");`

Output:

```
Enter the number of elements in the list 5
Enter the item?1
Enter the item?2
Enter the item?3
Enter the item?4
Enter the item?5
printing the list items....
1 2 3 4 5
```

Removing elements from the list

1. `List = [0,1,2,3,4]`
2. `print("printing original list: ");`
3. `for i in List:`
4. `print(i,end=" ")`
5. `List.remove(0)`
6. `print("\nprinting the list after the removal of first element...")`

7. **for i in** List:
8. **print**(i,end=" ")

Output:

printing original list:

0 1 2 3 4

printing the list after the removal of first element...

1 2 3 4

Python List Built-in functions

Python provides the following built-in functions which can be used with the lists.

SN	Function	Description
1	cmp(list1, list2)	It compares the elements of both the lists.
2	len(list)	It is used to calculate the length of the list.
3	max(list)	It returns the maximum element of the list.
4	min(list)	It returns the minimum element of the list.
5	list(seq)	It converts any sequence to the list.

Python List built-in methods

SN	Function	Description
1	<u>list.append(obj)</u>	The element represented by the object obj is added to the list.
2	<u>list.clear()</u>	It removes all the elements from the list.
3	<u>List.copy()</u>	It returns a shallow copy of the list.

4	<u>list.count(obj)</u>	It returns the number of occurrences of the specified object in the list.
5	<u>list.extend(seq)</u>	The sequence represented by the object seq is extended to the list.
6	<u>list.index(obj)</u>	It returns the lowest index in the list that object appears.
7	<u>list.insert(index, obj)</u>	The object is inserted into the list at the specified index.
8	<u>list.pop(obj=list[-1])</u>	It removes and returns the last object of the list.
9	<u>list.remove(obj)</u>	It removes the specified object from the list.
10	<u>list.reverse()</u>	It reverses the list.
11	<u>list.sort([func])</u>	It sorts the list by using the specified compare function if given.

Python Tuple

Python Tuple is used to store the sequence of immutable python objects.

Tuple is similar to lists since the value of the items stored in the list can be changed whereas the tuple is immutable and the value of the items stored in the tuple can not be changed.

A tuple can be written as the collection of comma-separated values enclosed with the small brackets. A tuple can be defined as follows.

1. T1 = (101, "Ayush", 22)
2. T2 = ("Apple", "Banana", "Orange")
3. tuple1 = (10, 20, 30, 40, 50, 60)
4. **print**(tuple1)
5. count = 0
6. **for** i **in** tuple1:
7. **print**("tuple1[%d] = %d"%(count, i));

Output:

```
(10, 20, 30, 40, 50, 60)
tuple1[0] = 10
tuple1[0] = 20
tuple1[0] = 30
tuple1[0] = 40
tuple1[0] = 50
tuple1[0] = 60
```

Example 2

1. `tuple1 = tuple(input("Enter the tuple elements ..."))`
2. `print(tuple1)`
3. `count = 0`
4. `for i in tuple1:`
5. `print("tuple1[%d] = %s"%(count, i));`

Output:

```
Enter the tuple elements ...12345
('1', '2', '3', '4', '5')
tuple1[0] = 1
tuple1[0] = 2
tuple1[0] = 3
tuple1[0] = 4
tuple1[0] = 5
```

Tuple indexing and splitting

The indexing and slicing in tuple are similar to lists. The indexing in the tuple starts from 0 and goes to `length(tuple) - 1`.

The items in the tuple can be accessed by using the slice operator. Python also allows us to use the colon operator to access multiple items in the tuple.

Consider the following image to understand the indexing and slicing in detail.

Tuple = (0, 1, 2, 3, 4, 5)					
0	1	2	3	4	5
Tuple[0] = 0		Tuple[0:] = (0, 1, 2, 3, 4, 5)			
Tuple[1] = 1		Tuple[:] = (0, 1, 2, 3, 4, 5)			
Tuple[2] = 2		Tuple[2:4] = (2, 3)			
Tuple[3] = 3		Tuple[1:3] = (1, 2)			
Tuple[4] = 4		Tuple[:4] = (0, 1, 2, 3)			
Tuple[5] = 5					

Unlike lists, the tuple items can not be deleted by using the del keyword as tuples are immutable. To delete an entire tuple, we can use the del keyword with the tuple name.

Consider the following example.

1. tuple1 = (1, 2, 3, 4, 5, 6)
2. **print**(tuple1)
3. **del** tuple1[0]
4. **print**(tuple1)
5. **del** tuple1
6. **print**(tuple1)

Output:

```
(1, 2, 3, 4, 5, 6)
```

```
Traceback (most recent call last):
```

```
File "tuple.py", line 4, in <module>
```

```
    print(tuple1)
```

```
NameError: name 'tuple1' is not defined
```

Basic Tuple operations

The operators like concatenation (+), repetition (*), Membership (in) works in the same way as they work with the list. Consider the following table for more detail.

Tuple t = (1, 2, 3, 4, 5) and Tuple t1 = (6, 7, 8, 9) are declared.

Operator	Description	Example
Repetition	The repetition operator enables the tuple elements to be repeated multiple times.	T1*2 = (1, 2, 3, 4, 5, 1, 2, 3, 4, 5)
Concatenation	It concatenates the tuple mentioned on either side of the operator.	T1+T2 = (1, 2, 3, 4, 5, 6, 7, 8, 9)
Membership	It returns true if a particular item exists in the tuple otherwise false.	print (2 in T1) prints True.
Iteration	The for loop is used to iterate over the tuple elements.	for i in T1: print(i) Output 1,2,3,4,5
Length	It is used to get the length of the tuple.	len(T1) = 5

Python Tuple inbuilt functions

SN	Function	Description
1	cmp(tuple1, tuple2)	It compares two tuples and returns true if tuple1 is greater than tuple2 otherwise false.

2	len(tuple)	It calculates the length of the tuple.
3	max(tuple)	It returns the maximum element of the tuple.
4	min(tuple)	It returns the minimum element of the tuple.
5	tuple(seq)	It converts the specified sequence to the tuple.

Where use tuple

Using tuple instead of list is used in the following scenario.

1. Using tuple instead of list gives us a clear idea that tuple data is constant and must not be changed.
 2. Tuple can simulate dictionary without keys. Consider the following nested structure which can be used as a dictionary.
1. [(101, "John", 22), (102, "Mike", 28), (103, "Dustin", 30)]
 3. Tuple can be used as the key inside dictionary due to its immutable nature.

List VS Tuple

SN	List	Tuple
1	The literal syntax of list is shown by the [].	The literal syntax of the tuple is shown by the ().
2	The List is mutable.	The tuple is immutable.

3	The List has the variable length.	The tuple has the fixed length.
4	The list provides more functionality than tuple.	The tuple provides less functionality than the list.
5	The list Is used in the scenario in which we need to store the simple collections with no constraints where the value of the items can be changed.	The tuple is used in the cases where we need to store the read-only collections i.e., the value of the items can not be changed. It can be used as the key inside the dictionary.

Nesting List and tuple

We can store list inside tuple or tuple inside the list up to any number of level.

```
Employees = [(101, "Ayush", 22), (102, "john", 29), (103, "james", 45),
(104, "Ben", 34)]
```

```
print("----Printing list----");
```

```
for i in Employees:
```

```
    print(i)
```

```
Employees[0] = (110, "David",22)
```

```
print();
```

```
print("-----Printing list after modification-----");
```

```
for i in Employees:
```

```
    print(i)
```

Output:

```
----Printing list----
```

```
(101, 'Ayush', 22)
```

```
(102, 'john', 29)
```

```
(103, 'james', 45)
```

```
(104, 'Ben', 34)
```

```
-----Printing list after modification-----
```

```
(110, 'David', 22)
(102, 'john', 29)
(103, 'james', 45)
(104, 'Ben', 34)
```

Python Set

The set in python can be defined as the unordered collection of various items enclosed within the curly braces. The elements of the set can not be duplicate. The elements of the python set must be immutable.

Creating a set

The set can be created by enclosing the comma separated items with the curly braces.

Example 1: using curly braces

```
Days = {"Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"}
```

```
print(Days)
```

```
print(type(Days))
```

```
print("looping through the set elements ... ")
```

1. **for** i **in** Days:
2. **print**(i)

Example 2: using set() method

1. Days = set(["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"])
2. **print**(Days)
3. **print**(type(Days))
4. **print**("looping through the set elements ... ")
5. **for** i **in** Days:
6. **print**(i)

Union of two Sets

The union of two sets are calculated by using the or (|) operator. The union of the two sets contains the all the items that are present in both the sets.

Example 1 : using union | operator

```
Days1 = {"Monday", "Tuesday", "Wednesday", "Thursday"}
```

```
Days2 = {"Friday", "Saturday", "Sunday"}
```

```
print(Days1|Days2) #printing the union of the sets
```

Output:

```
{'Friday', 'Sunday', 'Saturday', 'Tuesday', 'Wednesday', 'Monday',  
'Thursday'}
```

Python also provides the **union()** method which can also be used to calculate the union of two sets. Consider the following example.

Using union() method

```
Days1 = {"Monday", "Tuesday", "Wednesday", "Thursday"}
```

```
Days2 = {"Friday", "Saturday", "Sunday"}
```

```
print(Days1.union(Days2)) #printing the union of the sets
```

Output:

```
{'Friday', 'Monday', 'Tuesday', 'Thursday', 'Wednesday', 'Sunday',  
'Saturday'}
```

Intersection of two sets

The & (intersection) operator is used to calculate the intersection of the two sets in python. The intersection of the two sets are given as the set of the elements that common in both sets.

Example 1: using & operator

```
1. set1 = {"Ayush", "John", "David", "Martin"}
```

```
2. set2 = {"Steve", "Milan", "David", "Martin"}
```

3. **print**(set1&set2) #prints the intersection of the two sets

Output:

```
{'Martin', 'David'}
```

Example 2: using intersection() method

1. set1 = {"Ayush","John", "David", "Martin"}
2. set2 = {"Steave","Milan","David", "Martin"}
3. **print**(set1.intersection(set2)) #prints the intersection of the two sets

Output:

```
{'Martin', 'David'}
```

Set comparisons

Python allows us to use the comparison operators i.e., <, >, <=, >=, == with the sets by using which we can check whether a set is subset, superset, or equivalent to other set. The boolean true or false is returned depending upon the items present inside the sets.

```
Days1 = {"Monday", "Tuesday", "Wednesday", "Thursday"}
```

```
Days2 = {"Monday", "Tuesday"}
```

```
Days3 = {"Monday", "Tuesday", "Friday"}
```

#Days1 is the superset of Days2 hence it will print true.

```
print (Days1>Days2)
```

#prints false since Days1 is not the subset of Days2

```
print (Days1<Days2)
```

#prints false since Days2 and Days3 are not equivalent

```
print (Days2 == Days3)
```

Output:

```
True  
False
```

False

Frozenset for the dictionary

If we pass the dictionary as the sequence inside the frozenset() method, it will take only the keys from the dictionary and returns a frozenset that contains the key of the dictionary as its elements.

```
Dictionary = {"Name":"John", "Country":"USA", "ID":101}
```

```
print(type(Dictionary))
```

```
Frozenset = frozenset(Dictionary); #Frozenset will contain the keys of the  
dictionary
```

```
print(type(Frozenset))
```

```
for i in Frozenset:
```

```
    print(i)
```

Output:

```
<class 'dict'>  
<class 'frozenset'>  
Name  
Country  
ID
```

Python Dictionary

Dictionary is used to implement the key-value pair in python. The dictionary is the data type in python which can simulate the real-life data arrangement where some specific value exists for some particular key.

Creating the dictionary

The dictionary can be created by using multiple key-value pairs enclosed with the small brackets () and separated by the colon (:). The collections of the key-value pairs are enclosed within the curly braces {}.

The syntax to define the dictionary is given below.

```
Dict = {"Name": "Ayush", "Age": 22}
```

In the above dictionary **Dict**, The keys **Name**, and **Age** are the string that is an immutable object.

Create a dictionary:

```
Employee = {"Name": "John", "Age": 29, "salary": 25000, "Company": "GOOGLE"}  
print(type(Employee))  
print("printing Employee data .... ")  
print(Employee)
```

Output

```
<class 'dict'>  
printing Employee data ....  
{'Age': 29, 'salary': 25000, 'Name': 'John', 'Company': 'GOOGLE'}
```

Accessing the dictionary values

We have discussed how the data can be accessed in the list and tuple by using the indexing.

However, the values can be accessed in the dictionary by using the keys as keys are unique in the dictionary.

```
Employee = {"Name": "John", "Age": 29, "salary": 25000, "Company": "GOOGLE"}  
print(type(Employee))  
print("printing Employee data .... ")  
print("Name : %s" %Employee["Name"])  
print("Age : %d" %Employee["Age"])  
print("Salary : %d" %Employee["salary"])
```

```
print("Company : %s" %Employee["Company"])
```

Output:

```
<class 'dict'>
printing Employee data ....
Name : John
Age : 29
Salary : 25000
Company : GOOGLE
```

Updating dictionary values

The dictionary is a mutable data type, and its values can be updated by using the specific keys.

```
Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
```

```
print(type(Employee))
```

```
print("printing Employee data .... ")
```

```
print(Employee)
```

```
print("Enter the details of the new employee....");
```

```
Employee["Name"] = input("Name: ");
```

```
Employee["Age"] = int(input("Age: "));
```

```
Employee["salary"] = int(input("Salary: "));
```

```
Employee["Company"] = input("Company:");
```

```
print("printing the new data");
```

```
print(Employee)
```

Output:

```
<class 'dict'>
printing Employee data ....
{'Name': 'John', 'salary': 25000, 'Company': 'GOOGLE', 'Age': 29}
Enter the details of the new employee....
Name: David
Age: 19
```

```
Salary: 8900
Company:JTP
printing the new data
{'Name': 'David', 'salary': 8900, 'Company': 'JTP', 'Age': 19}
```

Deleting elements using del keyword

The items of the dictionary can be deleted by using the del keyword as given below.

```
Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
print(type(Employee))
print("printing Employee data .... ")
print(Employee)
print("Deleting some of the employee data")
del Employee["Name"]
del Employee["Company"]
print("printing the modified information ")
print(Employee)
print("Deleting the dictionary: Employee");
del Employee
print("Lets try to print it again ");
print(Employee)
```

Output:

```
<class 'dict'>
printing Employee data ....
{'Age': 29, 'Company': 'GOOGLE', 'Name': 'John', 'salary': 25000}
Deleting some of the employee data
printing the modified information
{'Age': 29, 'salary': 25000}
```


Deleting the dictionary: Employee

Lets try to print it again

Traceback (most recent call last):

```
File "list.py", line 13, in <module>
```

```
    print(Employee)
```

```
NameError: name 'Employee' is not defined
```

Python Functions

Functions are the most important aspect of an application. A function can be defined as the organized block of reusable code which can be called whenever required.

Python allows us to divide a large program into the basic building blocks known as function. The function contains the set of programming statements enclosed by {}. A function can be called multiple times to provide reusability and modularity to the python program.

Advantage of functions in python

There are the following advantages of C functions.

- By using functions, we can avoid rewriting same logic/code again and again in a program.
- We can call python functions any number of times in a program and from any place in a program.
- We can track a large python program easily when it is divided into multiple functions.
- Reusability is the main achievement of python functions.

Creating a function

In python, we can use **def** keyword to define the function. The syntax to define a function in python is given below.

```
def my_function():  
    function-suite
```

return <expression>

Function calling:

In python, a function must be defined before the function calling otherwise the python interpreter gives an error. Once the function is defined, we can call it from another function or the python prompt. To call the function, use the function name followed by the parentheses.

```
def hello_world():  
    print("hello world")
```

```
hello_world()
```

Output:

hello world

Parameters in function

The information into the functions can be passed as the parameters. The parameters are specified in the parentheses. We can give any number of parameters, but we have to separate them with a comma.

Example 1

#defining the function

```
def func (name):  
    print("Hi ",name);
```

#calling the function

```
func("Ayush")
```

Call by reference in Python

In python, all the functions are called by reference, i.e., all the changes made to the reference inside the function revert back to the original value referred by the reference.

Example 1 Passing Immutable Object (List)

```
1. #defining the function
2. def change_list(list1):
3.     list1.append(20);
4.     list1.append(30);
5.     print("list inside function = ",list1)
6.
7. #defining the list
8. list1 = [10,30,40,50]
9.
10.    #calling the function
11.    change_list(list1);
12.    print("list outside function = ",list1);
```

Output:

```
list inside function = [10, 30, 40, 50, 20, 30]
list outside function = [10, 30, 40, 50, 20, 30]
```

Types of arguments

There may be several types of arguments which can be passed at the time of function calling.

Required arguments

Keyword arguments

Default arguments

Variable-length arguments

Required Arguments

we can provide the arguments at the time of function calling.

Example 1

```
#the argument name is the required argument to the function func
def func(name):
```

```
message = "Hi "+name;
return message;
name = input("Enter the name?")
print(func(name))
```

Output:

```
Enter the name?John
Hi John
```

Keyword arguments

Python allows us to call the function with the keyword arguments. This kind of function call will enable us to pass the arguments in the random order.

Example 1

#function func is called with the name and message as the keyword arguments

```
def func(name,message):
    print("printing the message with",name,"and ",message)
func(name = "John",message="hello") #name and message is copied with the values John and hello respectively
```

Output:

```
printing the message with John and hello
```

Default Arguments

Python allows us to initialize the arguments at the function definition. If the value of any of the argument is not provided at the time of function call, then that argument can be initialized with the value given in the definition even if the argument is not specified at the function call.

Example 1

```
def printme(name,age=22):  
    print("My name is",name,"and age is",age)  
    printme(name = "john") #the variable age is not passed into the functi  
on however the default value of age is considered in the function
```

Output:

My name is john and age is 22

variable length Arguments

In the large projects, sometimes we may not know the number of arguments to be passed in advance. In such cases, Python provides us the flexibility to provide the comma separated values which are internally treated as tuples at the function call.

However, at the function definition, we have to define the variable with * (star) as * <variable - name >.

Example

```
def printme(*names):  
    print("type of passed argument is ",type(names))  
    print("printing the passed arguments...")  
    for name in names:  
        print(name)  
printme("john","David","smith","nick")
```

Output:

type of passed argument is
printing the passed arguments...
john
David
smith

nick

Python Lambda Functions

Python allows us to not declare the function in the standard manner, i.e., by using the `def` keyword. Rather, the anonymous functions are declared by using `lambda` keyword. Lambda functions can accept any number of arguments, but they can return only one value in the form of expression.

The syntax to define an Anonymous function is given below.

lambda arguments : expression

Example 1

```
x = lambda a:a+10 # a is an argument and a+10 is an expression which  
got evaluated and returned.  
print("sum = ",x(20))
```

Output:

```
sum = 30
```

Why use lambda functions?

The main role of the lambda function is better described in the scenarios when we use them anonymously inside another function. In python, the lambda function can be used as an argument to the higher order functions as arguments.

Example 1

#the function `table(n)` prints the table of `n`

```
def table(n):
```

```
    return lambda a:a*n; # a will contain the iteration variable i and a mu  
ltiple of n is returned at each function call
```

```
n = int(input("Enter the number?"))
```

```
b = table(n) #the entered number is passed into the function table. b will  
contain a lambda function which is called again and again with the iteratio  
n variable i
```

```
for i in range(1,11):
```

`print(n,"X",i,"=",b(i));` #the lambda function b is called with the iteration variable i,

Python File Handling

Till now, we were taking the input from the console and writing it back to the console to interact with the user.

Opening a file

Python provides the `open()` function which accepts two arguments, file name and access mode in which the file is accessed. The function returns a file object which can be used to perform various operations like reading, writing, etc.

The syntax to use the `open()` function is given below.

1. file object = `open(<file-name>, <access-mode>, <buffering>)`

SN	Access mode	Description
1	r	It opens the file to read-only. The file pointer exists at the beginning. The file is by default open in this mode if no access mode is passed.
2	rb	It opens the file to read only in binary format. The file pointer exists at the beginning of the file.
3	r+	It opens the file to read and write both. The file pointer exists at the beginning of the file.
4	rb+	It opens the file to read and write both in binary format. The file pointer exists at the beginning of the file.

5	w	It opens the file to write only. It overwrites the file if previously exists or creates a new one if no file exists with the same name. The file pointer exists at the beginning of the file.
6	wb	It opens the file to write only in binary format. It overwrites the file if it exists previously or creates a new one if no file exists with the same name. The file pointer exists at the beginning of the file.
7	w+	It opens the file to write and read both. It is different from r+ in the sense that it overwrites the previous file if one exists whereas r+ doesn't overwrite the previously written file.
8	wb+	It opens the file to write and read both in binary format. The file pointer exists at the beginning of the file.
9	a	It opens the file in the append mode. The file pointer exists at the end of the previously written file if exists any.
10	ab	It opens the file in the append mode in binary format. The pointer exists at the end of the previously written file. It creates a new file in binary format if no file exists with the same name.
11	a+	It opens a file to append and read both.
12	ab+	It opens a file to append and read both in binary format.

Example

```
#opens the file file.txt in read mode
```

```
fileptr = open("file.txt","r")
```

```
if fileptr:
```

```
    print("file is opened successfully")
```

Output:

```
<class '_io.TextIOWrapper'>
```

```
file is opened successfully
```

The close() method

Once all the operations are done on the file, we must close it through our python script using the close() method. Any unwritten information gets destroyed once the close() method is called on a file object.

The syntax to use the close() method is given below.

```
fileobject.close()
```

Example

```
# opens the file file.txt in read mode
```

```
fileptr = open("file.txt","r")
```

```
if fileptr:
```

```
    print("file is opened successfully")
```

```
    #closes the opened file
```

```
fileptr.close()
```

Reading the file

To read a file using the python script, the python provides us the read() method. The read() method reads a string from the file. It can read the data in the text as well as binary format.

The syntax of the read() method is given below.

```
fileobj.read(<count>)
```

Example

```
#open the file.txt in read mode. causes error if no such file exists.
```

```
fileptr = open("file.txt", "r");
```

```
#stores all the data of the file into the variable content
```

```
content = fileptr.read(9);
```

```
# prints the type of the data stored in the file
```

```
print(type(content))
```

```
#prints the content of the file
```

```
print(content)
```

```
#closes the opened file
```

```
fileptr.close()
```

Output:

```
<class 'str'>
```

```
Hi, I am
```

Looping through the file

By looping through the lines of the file, we can read the whole file.

Example

```
#open the file.txt in read mode. causes an error if no such file exists.
```

```
fileptr = open("file.txt", "r");
```

```
#running a for loop
```

```
for i in fileptr:
```

```
    print(i) # i contains each line of the file
```

Output:

Hi, I am the file and being used as
an example to read a
file in python.

Writing the file

To write some text to a file, we need to open the file using the open method with one of the following access modes.

a: It will append the existing file. The file pointer is at the end of the file. It creates a new file if no file exists.

w: It will overwrite the file if any file exists. The file pointer is at the beginning of the file.

Example 1

```
#open the file.txt in append mode. Creates a new file if no such file exists
. fileptr = open("file.txt","a");
  #appending the content to the file
fileptr.write("Python is the modern day language. It makes things so simple.")
  #closing the opened file
fileptr.close();
```

Now, we can see that the content of the file is modified.

File.txt:

Hi, I am the file **and** being used as
an example to read a
file **in** python.
Python **is** the modern day language. It makes things so simple.

Creating a new file

The new file can be created by using one of the following access modes with the function `open()`. **x**: it creates a new file with the specified name. It causes an error a file exists with the same name.

Example

1. `#open the file.txt in read mode. causes error if no such file exists.`
2. `fileptr = open("file2.txt","x");`
- 3.
4. `print(fileptr)`
- 5.
6. `if fileptr:`
7. `print("File created successfully");`

Output:

File created successfully

File Pointer positions

Python provides the `tell()` method which is used to print the byte number at which the file pointer exists. Consider the following example.

Example

```
# open the file file2.txt in read mode
fileptr = open("file2.txt","r")
#initially the filepointer is at 0
print("The filepointer is at byte :",fileptr.tell())
#reading the content of the file
content = fileptr.read();
#after the read operation file pointer modifies. tell() returns the location
of the fileptr.
print("After reading, the filepointer is at:",fileptr.tell())
```

Output:

The file pointer is at byte : 0

After reading, the file pointer is at 26

Modifying file pointer position

In the real-world applications, sometimes we need to change the file pointer location externally since we may need to read or write the content at various locations.

The syntax to use the seek() method is given below.

```
<file-ptr>.seek(offset[, from])
```

The seek() method accepts two parameters:

offset: It refers to the new position of the file pointer within the file.

from: It indicates the reference position from where the bytes are to be moved.

If it is set to 0, the beginning of the file is used as the reference position.
If it is set to 1, the current position of the file pointer is used as the reference position.

If it is set to 2, the end of the file pointer is used as the reference position.

Example

```
# open the file file2.txt in read mode
fileptr = open("file2.txt","r")

#initially the filepointer is at 0
print("The filepointer is at byte :",fileptr.tell())

#changing the file pointer location to 10.
fileptr.seek(10);
```

`#tell()` returns the location of the fileptr.

```
print("After reading, the filepointer is at:",fileptr.tell())
```

Output:

The file pointer is at byte : 0

After reading, the file pointer is at 10

Removing the file

The os module provides us the remove() method which is used to remove the specified file. The syntax to use the remove() method is given below.

```
remove(?file-name?)
```

Creating the new directory

The mkdir() method is used to create the directories in the current working directory. The syntax to create the new directory is given below.

```
mkdir(?directory name?)
```

Example

```
import os;
```

`#creating a new directory with the name new`

```
os.mkdir("new")
```

Python Modules

A python module can be defined as a python program file which contains a python code including python functions, class, or variables. In other words, we can say that our python code file saved with the extension (.py) is treated as the module. We may have a runnable code inside the python module.

Example

Create a module named as file.py which contains a function func that contains a code to print some message on the console.

Let's create the module named as **file.py**.

#displayMsg prints a message to the name being passed.

```
def displayMsg(name)
    print("Hi "+name);
```

Loading the module in our python code

We need to load the module in our python code to use its functionality. Python provides two types of statements as defined below.

1. The import statement
2. The from-import statement

The import statement

The import statement is used to import all the functionality of one module into another. Here, we must notice that we can use the functionality of any python source file by importing that file as the module into another python source file.

The syntax to use the import statement is given below.

```
import module1,module2,..... module n
```

Example:

```
import file;
name = input("Enter the name?")
file.displayMsg(name)
```

Output:

```
Enter the name?John
```

Hi John

The from-import statement

Instead of importing the whole module into the namespace, python provides the flexibility to import only the specific attributes of a module. This can be done by using from? import statement. The syntax to use the from-import statement is given below.

```
from < module-name> import <name 1>, <name 2>..,<name n>
```

calculation.py:

```
#place the code in the calculation.py
```

```
def summation(a,b):
```

```
    return a+b
```

```
def multiplication(a,b):
```

```
    return a*b;
```

```
def divide(a,b):
```

```
    return a/b;
```

Main.py:

```
from calculation import summation
```

```
#it will import only the summation() from calculation.py
```

```
a = int(input("Enter the first number"))
```

```
b = int(input("Enter the second number"))
```

```
print("Sum = ",summation(a,b)) #we do not need to specify the module  
name while accessing summation()
```

Output:


```
Enter the first number10
Enter the second number20
Sum = 30
```

Renaming a module

Python provides us the flexibility to import some module with a specific name so that we can use this name to use that module in our python source file.

```
import <module-name> as <specific-name>
```

Example

#the module calculation of previous example is imported in this example as cal.

```
import calculation as cal;
a = int(input("Enter a?"));
b = int(input("Enter b?"));
print("Sum = ",cal.summation(a,b))
```

Output:

```
Enter a?10
Enter b?20
Sum = 30
```

Using dir() function

The dir() function returns a sorted list of names defined in the passed module. This list contains all the sub-modules, variables and functions defined in this module.

Example

```
import json
List = dir(json)
print(List)
```

Output:

```
['JSONDecoder', 'JSONEncoder', '__all__', '__author__', '__builtins__',  
 '__cached__', '__doc__',  
 '__file__', '__loader__', '__name__', '__package__', '__path__',  
 '__spec__', '__version__',  
 '_default_decoder', '_default_encoder', 'decoder', 'dump', 'dumps',  
 'encoder', 'load', 'loads', 'scanner']
```

Python packages

The packages facilitate the developer with the application development environment by providing a **hierarchical directory structure**.

where a package contains sub-packages, modules, and sub-modules. The packages are used to **categorize the application level code efficiently**.

create a package named Employees in your home directory. Consider the following steps.

1. Create a directory with name Employees on path **/home**.
2. Create a python source file with name ITEmployees.py on the path **/home/Employees**.

ITEmployees.py

```
def getITNames():  
    List = ["John", "David", "Nick", "Martin"]  
    return List;
```

__init__.py

```
from ITEmployees import getITNames  
from BPOEmployees import getBPONames
```

Now, the directory **Employees** has become the package containing two python modules. Here we must notice that we must have to create `__init__.py` inside a directory to convert this directory to a package.

To use the modules defined inside the package Employees, we must have to import this in our python source file. Let's create a simple python source file at our home directory (/home) which uses the modules defined in this package.

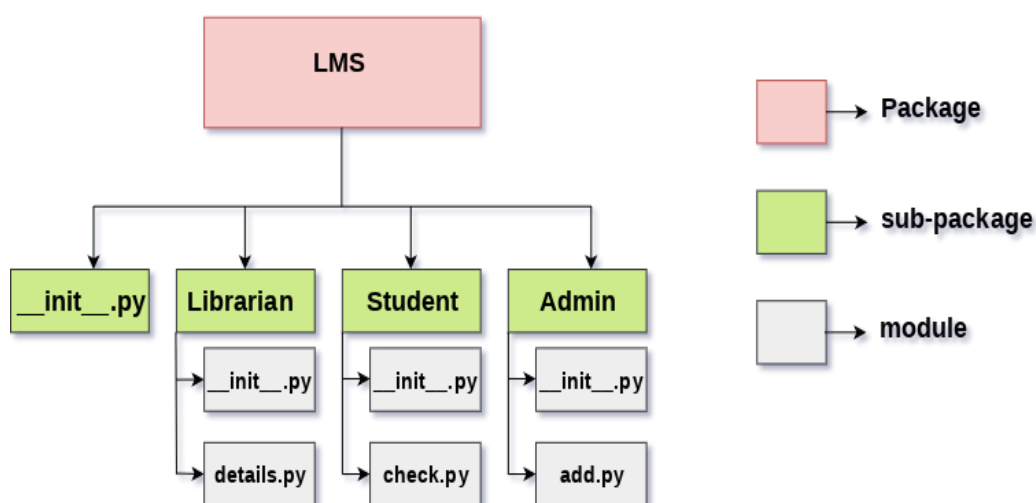
Test.py

```
import Employees
print(Employees.getNames())
```

Output:

```
['John', 'David', 'Nick', 'Martin']
```

The following image shows the directory structure of an application Library management system which contains three sub-packages as Admin, Librarian, and Student. The sub-packages contain the python modules.



Python Exceptions:

An exception can be defined as an abnormal condition in a program resulting disturb the flow of the program.

Whenever an exception occurs, the program **halts the execution**, and thus the **further code is not executed**. Therefore, an **exception is the error** which python script is unable to tackle with.

Common Exceptions

A list of common exceptions that can be thrown from a normal python program is given below.

ZeroDivisionError: Occurs when a **number is divided by zero**.

NameError: It occurs when a **name is not found**. It may be local or global.

Indentation Error: If incorrect indentation is given.

IOError: It occurs when Input Output operation fails.

EOFError: It occurs when the end of the file is reached, and yet operations are being performed.

Problem without handling exceptions

The exception is an abnormal condition that halts the execution of the program.

Example

```
a = int(input("Enter a:"))
b = int(input("Enter b:"))
c = a/b;
print("a/b = %d"%c)

#other code:
print("Hi I am other part of the program")
```

Output:

Enter a:10

Enter b:0

Traceback (most recent call last):

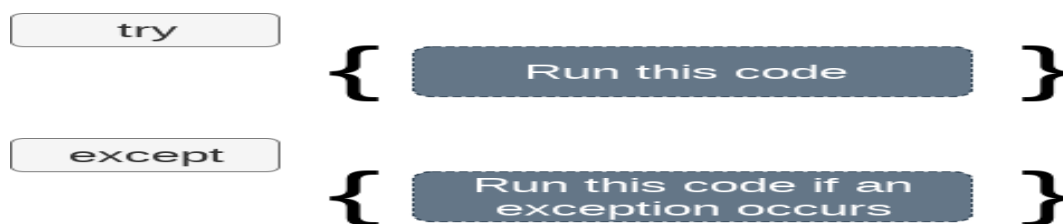
File "exception-test.py", line 3, in <module>

c = a/b;

ZeroDivisionError: division by zero

Exception handling in python

Python program place that code in the try block. The try block must be followed with the except statement which contains a block of code that will be executed if there is some exception in the try block.



SYNTAX:

try:

#block of code

except Exception1:

#block of code

except Exception2:

#block of code

#other code

The syntax to use the else statement with the try-except statement is given below.

try:

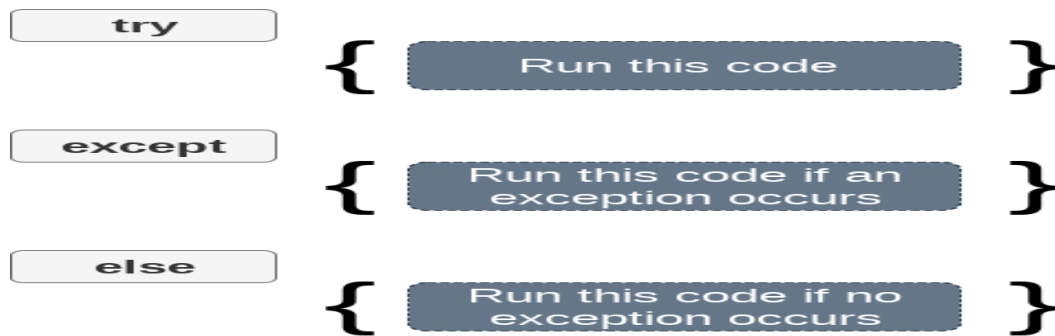
#block of code

except Exception1:

#block of code

else:

#this code executes if no except block is executed



Example

try:

a = int(input("Enter a:"))

b = int(input("Enter b:"))

c = a/b;

print("a/b = %d"%c)

except Exception:

print("can't divide by zero")

else:

print("Hi I am else block")

Output:

Enter a:10

Enter b:2

a/b = 5

Hi I am else block

The except statement with no exception

Python provides the flexibility not to specify the name of exception with the except statement.

Consider the following example.

Example

try:

```
a = int(input("Enter a:"))  
b = int(input("Enter b:"))  
c = a/b;  
print("a/b = %d"%c)
```

except:

```
print("can't divide by zero")  
else:  
print("Hi I am else block")
```

Output:

Enter a:10

Enter b:0

can't divide by zero

The finally block

We can use the finally block with the try block in which, we can place the important code which must be executed before the try statement throws an exception.

The syntax to use the finally block is given below.

Syntax

try:

```
# block of code  
# this may throw an exception
```

finally:

```
# block of code  
# this will always be executed
```

Raising exceptions

An exception can be raised by using the raise clause in python. The syntax to use the raise statement is given below.

syntax

1. **raise** Exception_class,<value>

Example

try:

```
age = int(input("Enter the age?"))
```

```
if age<18:
```

```
    raise ValueError;
```

```
else:
```

```
    print("the age is valid")
```

except ValueError:

```
    print("The age is not valid")
```

Output:

Enter the age?17

The age is not valid

Python Date and time

In python, the date is not a data type, but we can work with the date objects by importing the module named with datetime, time, and calendar.

Tick

In python, the time instants are counted since 12 AM, 1st January 1970. The function `time()` of the module time returns the total number of ticks

spent since 12 AM, 1st January 1970. A tick can be seen as the smallest unit to measure the time.

Example

```
import time;
#prints the number of ticks spent since 12 AM, 1st January 1970
print(time.time())
```

Output:1545124460.9151757

How to get the current time?

The localtime() functions of the time module are used to get the current time tuple. Consider the following example.

Example

```
import time;

#returns a time tuple

print(time.localtime(time.time()))
```

Output:

```
time.struct_time(tm_year=2018,      tm_mon=12,      tm_mday=18,
tm_hour=15, tm_min=1,
tm_sec=32, tm_wday=1, tm_yday=352, tm_isdst=0)
```

Python sleep time

The sleep() method of time module is used to stop the execution of the script for a given amount of time. The output will be delayed for the number of seconds given as float.

Example

1. **import** time
2. **for** i **in** range(0,5):
3. **print**(i)
4. #Each element will be printed after 1 second
5. time.sleep(1)

Output: 0,1,2,3,4

The datetime Module

The datetime module enables us to create the custom date objects, perform various operations on dates like the comparison, etc.

To work with dates as date objects, we have to import datetime module into the python source code.

Example

1. **import** datetime;
 #returns the current datetime object
 print(datetime.datetime.now())

Output:

```
2018-12-18 16:16:45.462778
```

Creating date objects

We can create the date objects by passing the desired date in the datetime constructor for which the date objects are to be created.

Example

```
import datetime;  
  
#returns the datetime object for the specified date  
  
print(datetime.datetime(2018,12,10))
```

Output:

```
2018-12-10 00:00:00
```

The calendar module

Python provides a calendar object that contains various methods to work with the calendars.

Example

```
import calendar;
cal = calendar.month(2018,12)
#printing the calendar of December 2018
print(cal)
```

Output:

```
javatpoint@localhost:~
File Edit View Search Terminal Help
[javatpoint@localhost ~]$ python3 time2.py
December 2018
Mo Tu We Th Fr Sa Su
                1  2
 3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31
javaTpoint
[javatpoint@localhost ~]$
```

Printing the calendar of whole year

The `prcal()` method of calendar module is used to print the calendar of the whole year. The year of which the calendar is to be printed must be passed into this method.

Example

```
import calendar
```

```
#printing the calendar of the year 2019  
calendar.prcal(2019)
```

Python Regular Expressions:

The regular expressions can be defined as the sequence of characters which are used to search for a pattern in a string.

The module `re` provides the support to use regex in the python program.

The `re` module throws an exception if there is some error while using the regular expression.

The **re** module must be imported to use the regex functionalities in python.

```
import re
```

Regex Functions

SN	Function	Description
1	match	This method matches the regex pattern in the string with the optional flag. It returns true if a match is found in the string otherwise it returns false.
2	search	This method returns the match object if there is a match found in the string.
3	find all	It returns a list that contains all the matches of a pattern in the string.
4	split	Returns a list in which the string has been split in each match.
5	sub	Replace one or many matches in the string.

Forming a regular expression

A regular expression can be formed by using the mix of meta-characters, special sequences, and sets.

Meta-Characters

Metacharacter is a character with the specified meaning.

Metacharacter	Description	Example
[]	It represents the set of characters.	"[a-z]"
\	It represents the special sequence.	"\r"
.	It signals that any character is present at some specific place.	"Ja.v."
^	It represents the pattern present at the beginning of the string.	"^Java"
\$	It represents the pattern present at the end of the string.	"point"
*	It represents zero or more occurrences of a pattern in the string.	"hello*"
+	It represents one or more occurrences of a pattern in the string.	"hello+"
{ }	The specified number of occurrences of a pattern the string.	"java{2}"
	It represents either this or that character is present.	"java point"
()	Capture and group	

Special Sequences

Special sequences are the sequences containing \ followed by one of the characters.

Character	Description
-----------	-------------

<code>\A</code>	It returns a match if the specified characters are present at the beginning of the string.
<code>\b</code>	It returns a match if the specified characters are present at the beginning or the end of the string.
<code>\B</code>	It returns a match if the specified characters are present at the beginning of the string but not at the end.
<code>\d</code>	It returns a match if the string contains digits [0-9].
<code>\D</code>	It returns a match if the string doesn't contain the digits [0-9].
<code>\s</code>	It returns a match if the string contains any white space character.
<code>\S</code>	It returns a match if the string doesn't contain any white space character.
<code>\w</code>	It returns a match if the string contains any word characters.
<code>\W</code>	It returns a match if the string doesn't contain any word.
<code>\Z</code>	Returns a match if the specified characters are at the end of the string.

The `findall()` function

This method returns a list containing a list of all matches of a pattern within the string. It returns the patterns in the order they are found. If there are no matches, then an empty list is returned.

Example

```
import re

str = "How are you. How is everything"

matches = re.findall("How", str)

print(matches)
```

```
print(matches)
```

Output:

```
['How', 'How']
```

The match object

The match object contains the information about the search and the output. If there is no match found, the None object is returned.

Example

```
import re
```

```
str = "How are you. How is everything"
```

```
matches = re.search("How", str)
```

```
print(type(matches))
```

```
print(matches) #matches is the search object
```

Output:

```
<class '_sre.SRE_Match'>  
<_sre.SRE_Match object; span=(0, 3), match='How'>
```