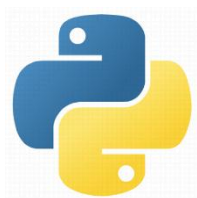




केन्द्रीय विद्यालय संगठन नई दिल्ली
Kendriya Vidyalaya Sangathan New Delhi

शिक्षा एवं प्रशिक्षण का आंचलिक संस्थान, भुवनेश्वर
Zonal Institute of Education & Training Bhubaneswar



COMPILED
**PYTHON MATERIAL
FOR COMPUTER SCIENCE AND
INFORMATICS PRACTICES**

CLASS XI

SESSION 2018-19

COMPILED DURING

3- Day workshop

“To Prepare Master Trainer for PGT (Computer Science)”

२५-२७ जून २०१८ / 25-27 June 2018

from

KVS Regions

Bhubaneswar, Ranchi, Kolkata, Guwahati, Tinsukia & Silchar

Chief Patron:

**Sh. A.V.L.J. Rao,
Director, ZIET, Bhubaneswar and
Deputy Commissioner
KVS ,RO, Bhubaneswar**

Co-ordinator:

**Dr. Abhijit Saha
Training Associate
ZIET Bhubaneswar**

Resource Persons:

**Mr. Manash Ranjan Sahoo
PGT (Comp. Sc.), K.V. No.4 Bhubaneswar(Bhubaneswar Region)**

**Mr. Dipayan Sarkar
PGT (Comp. Sc.), K.V. Gumla (Ranchi Region)**

Foreword

It gives me immense pleasure to introduce this booklet “Python Material for Computer Science & Informatics Practices for Class XI” to all readers.



This study material has been prepared by the teachers who participated in the 3 Day workshop to prepare Master Trainers for PGT (Computer Science) during 25-27 June 2018 at ZIET Bhubaneswar. It is the result of the sincere and hard work put in by the Resource persons, Guest faculties, participants and the co-ordinator during these three days is the workshop.

Session 2018-19 onwards Kendriya Vidyalaya Sangathan has adopted Python Programming in class XI in place of C++. It was thus essential to provide training to all the PGT (Computer Science) of KVS and make them aware of the changed curriculum. Accordingly a 3 day workshop was planned by the KVS HQ and ZIETs were assigned the task to prepare master trainers. These trainers would further provide training to other teachers of their respective regions.

ZIET Bhubaneswar conducted the workshop accordingly and came out with a study material which would be useful for teachers and the taught. All readers are requested to go through the booklet and give their valuable feedback to us. You are free to give suggestions for improvement of this booklet which will be helpful to bring out a revised edition in due course.

Please forward this booklet to every concerned teacher and students who are studying the subject.

My best wishes to every reader. Thanks to all contributors for this booklet. I hope this booklet will help you to cope with the new curriculum.

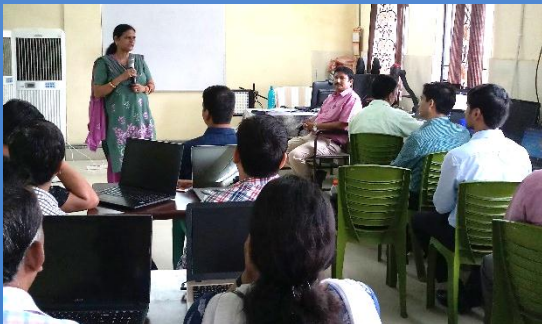
A handwritten signature in blue ink, appearing to be 'AVLJ Rao', with a long horizontal line extending to the right.

(AVLJ Rao)

Director, ZIET Bhubaneswar &
Deputy Commissioner, KVS Bhubaneswar Region

Subject Contributors

Sl.No	Name of the PGT(Computer Science)	Name of the K.V.	Name of the KVS Region
1	Mr. Narendra Kumar	Silchar	Silchar
2	Mr. Himanshu Khare	No.1, Kunjaban	Silchar
3	Mr. Abhishek Arya	Happy Valley Shillong	Silchar
4	Mr. Swapan Kumar Malakar	Ballygunge	Kolkata
5	Mrs. MADHULIKA DEBNATH	AFS Barrackpore	Kolkata
6	Mr. Amit Kumar Malla	Command Hospital	Kolkata
7	Mrs. Pournomi Sen	OF Dum Dum	Kolkata
8	Mr. AMARENDRA KUMAR JHA	No.2 Salt Lake	Kolkata
9	Mrs. NANDINI DAS	No.2 RS Kharagpur	Kolkata
10	Mr. ARBIND KUMAR JHA	No.1 Salt Lake	Kolkata
11	Mrs. Sabiha Shahin	AFS Bagdogra	Kolkata
12	Mr. MANOJ KUMAR MISHRA	No.1 Baripada	BHUBANESWAR
13	Mr. V.K. MEHRA	Gopalpur Mil. Stn.	BHUBANESWAR
14	Mr.S.K. Behera	Bhawanipatna	BHUBANESWAR
15	Ms. SEEMA DEVI	ARC Charbatia	BHUBANESWAR
16	Mr.Tanmaya Mishra	Puri	BHUBANESWAR
17	Mr. SANATAN BANJI	No.1 Sambalpur	BHUBANESWAR
18	Mr. Vaibhav Jain	Rayagada	BHUBANESWAR
19	Mr. Rajendra Kumar Sahu	No.2, CRPF, Bhubaneswar	BHUBANESWAR
20	Mr. P K Lohani	Duliajan	Tinsukia
21	Mr. Sumit Kumar Chaudhary	Tinsukia	Tinsukia
22	Mr. Deepak Kumar Gupta	No.1 Imphal	Tinsukia
23	Ms. Bhuvnesh Kumari	NERIST (NIRJULI)	Tinsukia
24	Ms. Suman	Dinjan	Tinsukia
25	Mrs. Gitanjali Sadangi	Tatanagar	Ranchi
26	Mr.R N P Sinha	Hinoo Ranchi (2nd Shift)	Ranchi
27	Mr. Pravin Kumar Singh	No. 1 Dhanbad	Ranchi
28	Mr. Amod Kumar Singh	Patratu	Ranchi
29	Mr. MANTOSH KUMAR	Dipatoli Ranchi	Ranchi
30	Mr. Girish Bhatt	Rangia	Guwahati
31	Ms. Simpa Kharagwanshi	Goalpara	Guwahati
32	Mr. Prem Chand	Diphu	Guwahati
33	Mr. Umed Ali	Missamari	Guwahati
34	Mr. Ramesh Kumar Kantha	Doomdooma	Guwahati
35	Mr. Vijay Shankar	Sivasagar(ONGC)	Guwahati
36	Mrs. Huma Yasmin	IOC Noonmati	Guwahati
37	Mr. Pawan Kumar	Jagiroad	Guwahati
38	Mr. Manish Kr. Prajapati	Borjhar	Guwahati



INTRODUCTION TO PYTHON:

Familiarization with Python Programming

A Python is a general purpose high level programming language. It is a platform independent programming language.

- It has very simple and straight forward syntax. Anyone can learn it easily and can be opted as the first programming language
- It is also a case sensitive language like C, C++ & Java.
- It is an Object Oriented Language like Java and C++, in fact every variable in Python is an Object.
- Python is a dynamically typed language as there is no need of compiler and it uses interpreter which dynamically decide the variable. Since it is an interpreted language it executes one statement or command at a time.
- We can use variable without declaration as it is automatically declared at a time
- Indentation is used in place of curly brackets which increase the readability of code.

Features-

- Emphasis on code readability.
- Automatic memory management like in Java.
- Dynamically typed.
- large library
- Multi-paradigm programming language (Object Oriented, procedural etc.)
- Python is interactive interpreter it is easy to check Python commands.
- Platform independent.

Python Library:

It has a huge predefined library, function or modules. Because of this extensive libraries, Python is popular among developers.

- Graphical user interfaces
- Automation
- Web frameworks
- Documentation
- Multimedia
- System administration
- Databases
- System computing
- Networking
- Text processing
- Test frameworks
- Image processing
- Web scraping (like crawler)
- IOT

It can be used for

- console application
- desktop application like calculator
- web application
- mobile application
- machine learning
- IOT things

Popular Apps developed in Python

YouTube, google, Dropbox, Quora, Instagram etc

DISADVANTAGES:

-execution speed is generally slower. For application involving large datasets, complex maths it is generally be efficient to use a compiled language rather than compiled language.

-Protecting code is difficult-because python is interpreted it is very difficult to hide code from prime eyes.

-python has design restriction because the language is dynamically typed, it requires more testing and has errors only show up at run time.

Installation of Python

To install Python, we must download the installation package of the required version from the following link/URL given below:

<https://www.python.org/downloads/>

To write and run Python programs interactively, we can either use the command line window or the IDLE.

IDLE is a simple Integrated Development Environment that comes with Python.

The most important feature of IDLE is that it is a program that allows the user to edit, run, browse and debug a Python program from a single interface.

STARTING IDLE

From Start Menu, open IDLE as follows:

Start menu--> Apps by name--> IDLE(Python 3.6 32 bit)

Or

Click on the icon to start IDLE

It always starts up in the shell.

Python IDLE comprises Python shell (Interactive mode) and Python Editor (Script mode).

Python shell is an interactive window where we can type Python code and see the output in the same window by pressing enter key. It is an interface between Python commands and the OS. Prompt of python ILDE is represent by (>>>).A user can write commands in front of command prompt (>>>) and see output by pressing enter key.

Python script mode is basically used for writing programs i.e. a set of instructions and can be saved for further use. A user can switch to script mode by creating a file from File→ New option and after writing instructions, user will save the newly written program with filename followed by a period (.) sign and py extension from File→ Save as option while saving the Python program first time.

Example File name: Helloworld.py

The output of program file can be viewed by executing the program from Run Menu→ Run Module or pressing F5 key.

Practical Implementation

First Python Program

Type the following code in any text editor or an IDE and save it as helloWorld.py

```
print ("Hello world!")
```

Now at the command window, go to the location of this file. You can use the cd command to change directory.

To run the script, type python helloWorld.py in the command window.

We should be able to see the **output** as follows:

```
Hello world!
```

MORE COMMANDS

```
-> print ("Welcome to World of Python programming")
```

```
Welcome to Python Programming
```

```
->print(20*55)
```

```
1100
```

```
->print(22/7)
```

```
3.142857142857143
```

Exiting Python

In order to exit from python command, click Ctrl + Z and press Enter key or type quit () or exit () function/statement and press Enter key.

Prerequisites:-

- Concept of variable in respect of python.
 - Concept of mutable/immutable datatype. Data types in Python

Every value in Python has a datatype. Since everything is an object in Python programming, data types are actually classes and variables are instance (object) of these classes like most other languages.

Python sets the variable type based on the value that is assigned to it. Unlike more languages, Python will change the variable type if the variable value is set to another value because a variable maintain an reference instead of a value itself. E.g

A=10 # will create an integer variable

A="Hello" # variable A is now String type

Python has five standard Data Types:

- Numbers
- String
- List
- Tuple
- Dictionary

Python Numbers

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

a=10

You can also delete the reference to a number object by using the del statement.

```
del a
```

or

```
del a, b # for deleting multiple reference.
```

After this statement such variable/s cannot be referred again.

Python supports four different numerical types –

- int (signed integers)
- long (long integers, they can also be represented in octal and hexadecimal)
- float (floating point real values)
 - complex (complex numbers) Most of the time using the standard Python number type is fine. Python will automatically convert a number from one type to another if it needs. But, under certain circumstances that a specific number type is needed (ie. complex, hexadecimal), the format can be forced into a format by using additional syntax in the table below:

Type Format Description

int	a = 10	Signed Integer
-----	--------	----------------

long a = 345L (Dropped since python 3.0. Use int instead)

float a = 45.67 (.) Floating point real values

complex a = 3.14J (J) Contains integer in the range 0 to 255.

Most of the time Python will do variable conversion automatically. You can also use Python conversion functions (int(), float(), complex()) to convert data from one type to another. We can also use type function to identify the datatype for the variable.

```
message = "Hello Python"
```

```
num = 1991
```

```
pi = 3.141
```

```
print(type(message)) # This will return a string
```

```
print(type(num)) # This will return an integer
```

```
print(type(pi)) # This will return a float
```

Integers can be of any length, it is only limited by the memory available.

A floating point number is accurate up to 15 decimal places. Integer and floating points are separated by decimal points. 1 is integer, 1.0 is floating point number. Complex numbers are written in the form, $x + yj$, where x is the real part and y is the imaginary part. Here are some examples.

```
c = 1+2j
```

Python Strings

Strings in Python are identified as a contiguous set of characters represented in the quotation marks. It is sequence of Unicode characters. String in python can be created using single quotes ' ', double quotes " ", or triple quotes """ """. String created using triple quotes only can go for multiple lines.

```
a= 'hello'
```

```
a="hello"
```

```
a= """ this is a  
multiline string..... """
```

Like other languages, String is indexed here and it's indexes starting at 0 in the beginning of the string and working their way from -1 at the end. String in python can also be accessed with negative indexed in reverse order.

+ operator can be used for concatenation and * is used for multiplication.

e.g.

```
str = 'Hello World!'
```

```
print (str) # Prints complete string
```

```
print (str[0]) # Prints first character of the string
```

```
print (str[2:5]) # Prints characters starting from 3rd to 5th
```

```
print (str[2:]) # Prints string starting from 3rd character
```

```
print (str * 2) # Prints string two times  
print (str + "TEST") # Prints concatenated string
```

OUTPUT:-

Hello World!

H

llo

llo World!

Hello World!Hello World!

Hello World!TEST

NOTE:- Python strings are "immutable" which means they cannot be changed after they are created (Java strings also use this immutable style). Since strings can't be changed, we construct *new* strings as we go to represent computed values. So for example the expression ('hello' + 'there') takes in the 2 strings 'hello' and 'there' and builds a new string 'hellothere'.

String Methods:-

Method Name	Purpose	Example
lower()	Converts string into lowercase	s.lower
upper()	Converts string into UPPERCASE	s.upper()
isalpha()		

tests if all the string chars are in characters only.

s.isalpha()

isdigit()

tests if all the string chars are in digit only

s.isdigit()

isspace()

tests if all the string chars are in spaces

s.isspace()

find() searches for the given other string (not a regular expression) within s, and returns the first index where it begins or -1 if not found

- s.find('other')

replace returns a string where all occurrences of 'old' have been replaced by 'new'

s.replace('old', 'new')

split() returns a list of substrings separated by the given delimiter

s.split('delim').

startswith() Check whether that string starts with a particular string

s.startswith('A')

endswith() Check whether that string ends with a particular string

s.endswith('A')

join() opposite of split(), joins the elements in the given list together using the string as the delimiter.

s.join(list)

strip() Removes whitespaces from start and end s.strip()

Introduce the notion of a variable, and methods to manipulate it (Concept of L-value and R-value even if not taught explicitly)

VARIABLE:

A variable in Python represents named location that refers to a value and whose values can be used and processed during program run.

In other words variables are named Labels, whose value can be used and processed during program run. So these are also known as Symbolic Variables.

Creating a Variable

Python variables are created by assigning value of desired type to them, e.g. to create a numeric variable, assign a numeric value to **variable_name;**

To create a string variable, assign a string value to variable_name and so on.

Example:

name='Mohan'
Type

Variable created of String

Age=15
Type)

Variable created of Numeric (Integer

Marks=59.5
(floating point Type)

Variable created of Numeric

Variables are Not Storage Containers in Python

Variable is a Container in traditional Programming Languages like C, C++ i.e., It is a named storage location that stores a value in it.

Example:

Num=15

Num=25

First the value 15 is assigned to variable **Num** and then value 25 is assigned to it. Suppose the variable **Num** is created as a container at a memory address say 10511 and it stores value as 15 in it. With the next statement the location of the variable did not change, only its contents changed.

15

25

Memory Address: 10511
10511

Memory Address:

But Python's Handling of Variables is Different:

Consider the same example as shown above:

Num=15

Num=25

15

25

Memory Address: 10511
10650

Memory Address:

Thus variables in Python do not have fixed locations unlike other programming languages. The location they refer to changes every time their values change. **But this Rule is not for all types of Variables.**

Lvalues and Rvalues:

Lvalues are the objects to which you can assign a value or expression. Lvalues can come on LHS or RHS of an assignment statement.

Rvalues are the literals and expressions that are assigned to lvalues. Rvalues can come on RHS of an assignment statement.

Example:

a=20

b=10

But the following statements are not valid and will produce an error:

20=a

or

10=b

Multiple Assignments: Python is very versatile with assignments.

1. Assigning same value to multiple variables

We can assign same value to multiple variables:

Example: a=b=c=10

2. Assigning Multiple values to multiple variables

We can assign multiple values to multiple variables in single statement e.g.

X, Y, Z=10, 20, 30

It will assign the values order wise, i.e. first variable is given first value, second variable the second value and so on.

This style of assigning values is very useful and compact.

Consider the code below:

```
X,Y=25,50
```

```
print(X,Y)
```

It will print result as

```
25 50
```

Now if you want to swap values of X and Y, you just need to write:

```
X,Y=Y,X
```

```
print(X,Y)
```

Now the result will be 50 25

Variable Definition

In Python, a variable is created when you first assign a value to it. It also means that a variable is not created until some value is assigned to it.

Consider the following code:

```
>>> print(x)
```

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

```
File "<pyshell#0>", line 1, in <module>
```

```
print(x)
```

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

When you run the above code, it will produce an error for the statement -name 'x' is not defined.

So to correct the above code, you need to first assign something to x before using it in a statement.

```
x=20
```

```
print(x)
```

```
20
```

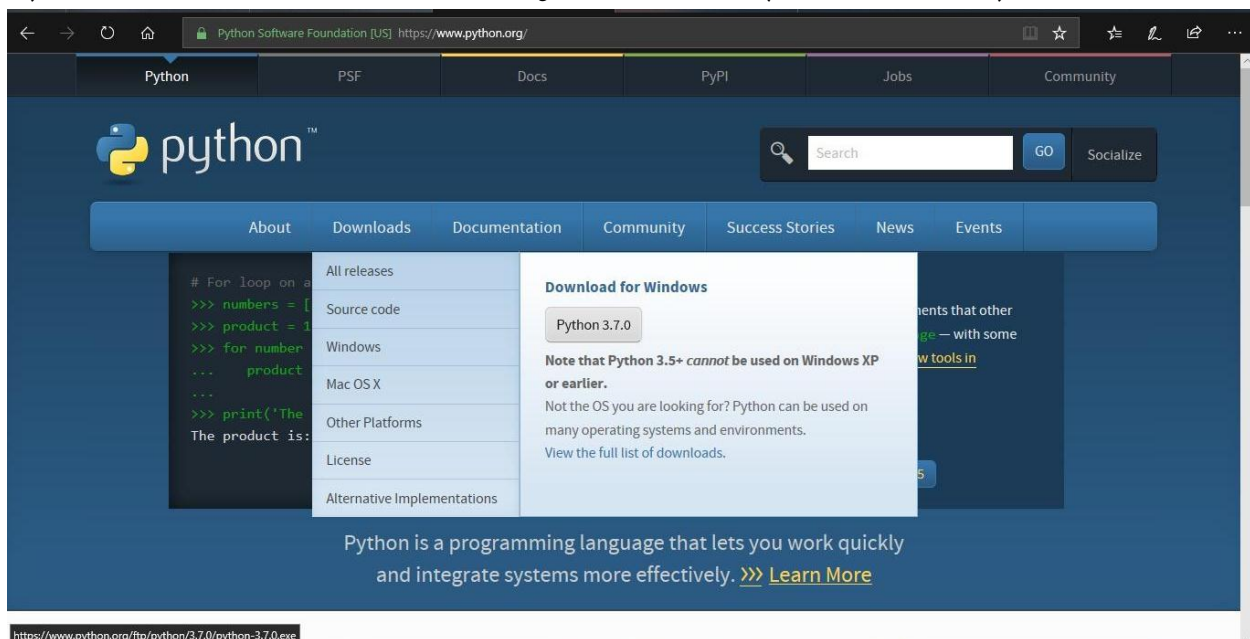
Accepting Input from the Console & Assignment Statement

Python is a programming language that lets you work quickly and integrate systems more effectively.

- 1) Python is an interpreted, interactive, object-oriented programming language.
- 2) It incorporates modules, exceptions, dynamic typing, very high level dynamic data types, and classes.
- 3) Python combines remarkable power with very clear syntax.
- 4) It has interfaces to many system calls and libraries, as well as to various window systems, and is extensible in C or C++.
- 5) It is also usable as an extension language for applications that need a programmable interface.
- 6) Finally, Python is portable: it runs on many Unix variants, on the Mac, and on Windows 2000 and later.

Downloading & Opening of Python :

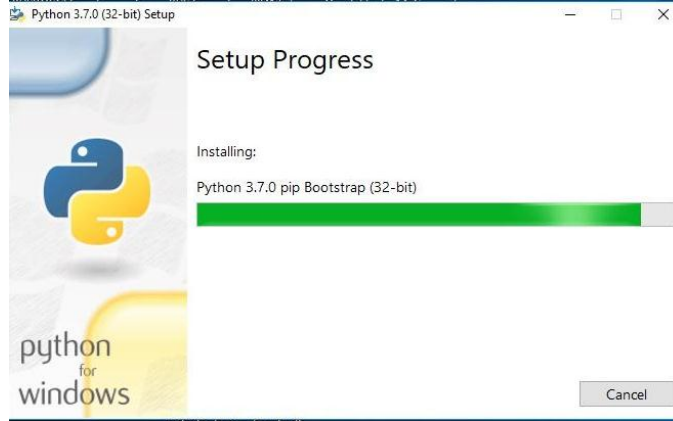
- 1) Log in to <https://www.python.org/>
- 2) Under “Downloads” click Python 3.7.0 (for Windows)



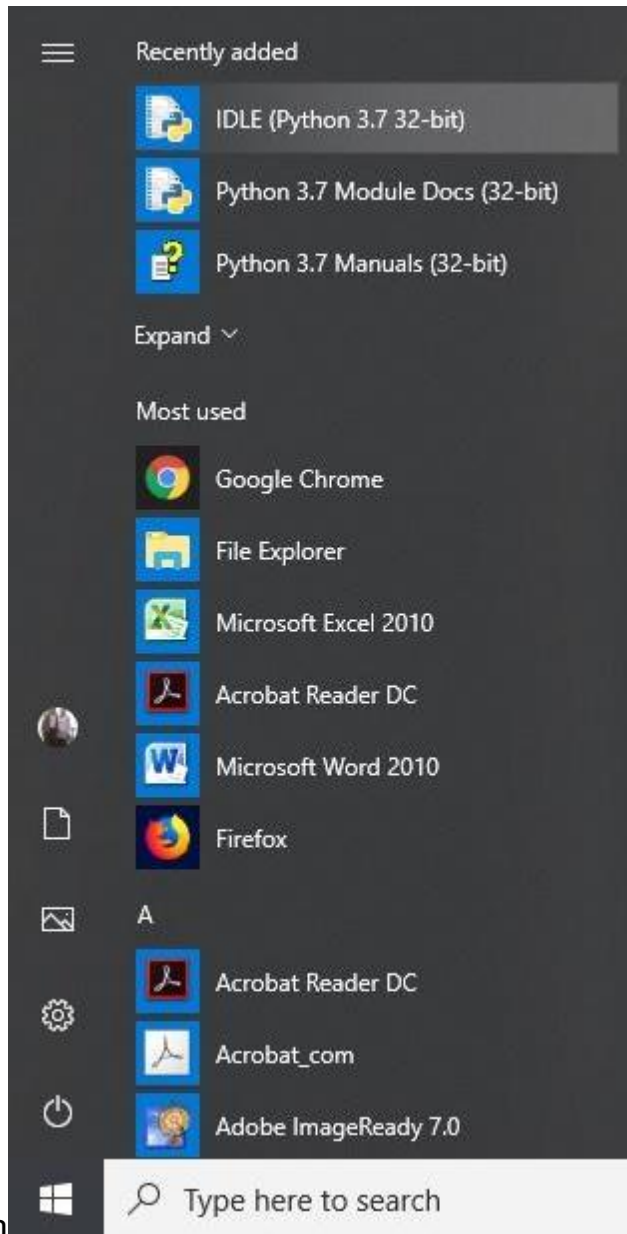
- 3) Then save the “python-3.7.0.exe” file.



- 4) After saving, run the exe file from the downloads and Setup Progress



5) Click on “**IDLE**(Python 3.7 32-bit)” under Programs to run the



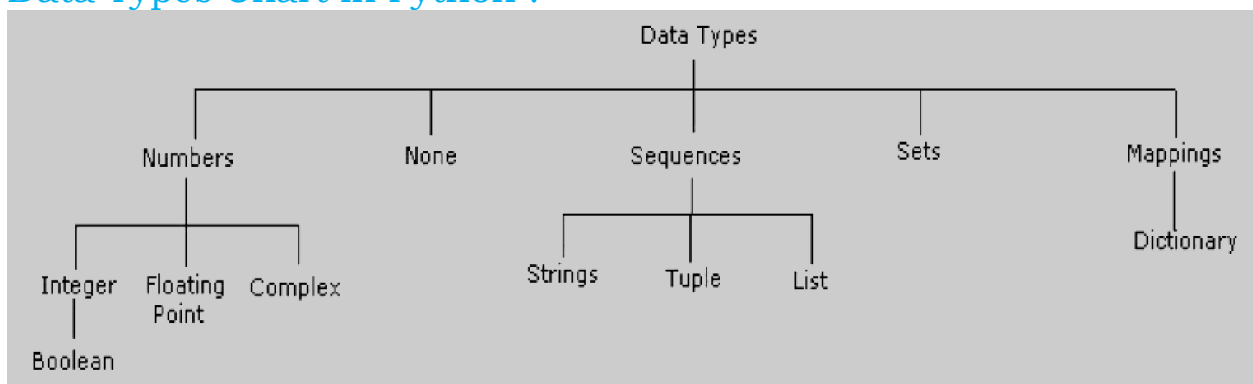
Python

6) As soon as it is clicked, Python Opens

```
Python 3.7.0 Shell
File Edit Shell Debug Options Window Help
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:06:47) [MSC v.1914 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
```

Ln: 3 Col: 4

Data Types Chart in Python :



Assignment Operator :

Symbol	Description
=	Assigned values from right side operands to left variable

Accepting Input from the Console :

A Program needs to interact with end user to accomplish the desired task, this is done by using Input-Output facility. Input means the data entered by the user (end user) of the program. While writing algorithm(s), getting input from user was represented as follows :

Built-in Function : `input()`

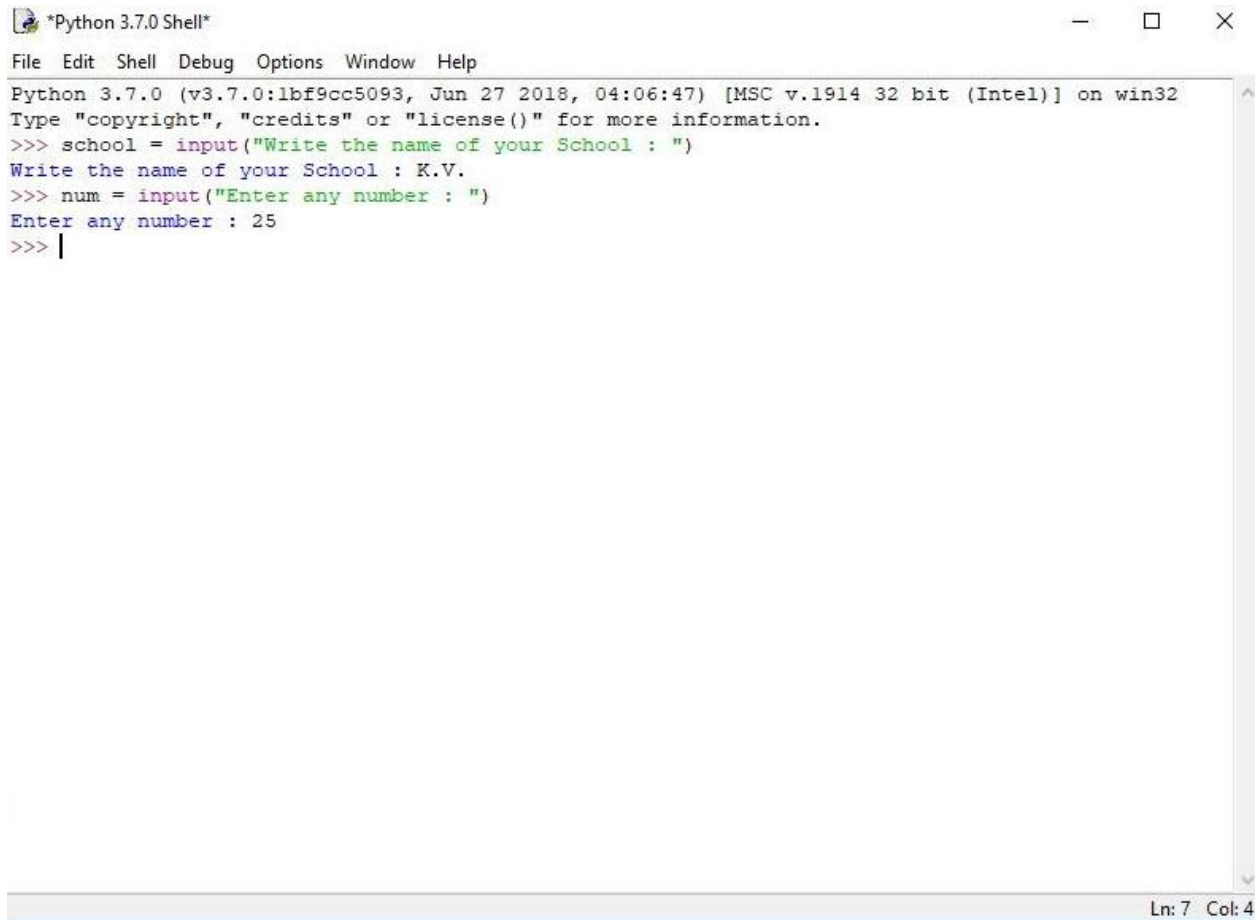
Syntax :

`variable_that_holds_the_value = input("<message to be displayed>")`

Example :

```
school = input("Write the name of your school : ")
```

```
num = input("Enter any number : ")
```

A screenshot of a Python 3.7.0 Shell window. The window title is "*Python 3.7.0 Shell*" and it has standard Windows window controls (minimize, maximize, close). The menu bar includes "File", "Edit", "Shell", "Debug", "Options", "Window", and "Help". The main area shows the following text:

```
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:06:47) [MSC v.1914 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> school = input("Write the name of your School : ")
Write the name of your School : K.V.
>>> num = input("Enter any number : ")
Enter any number : 25
>>> |
```

The status bar at the bottom right indicates "Ln: 7 Col: 4".

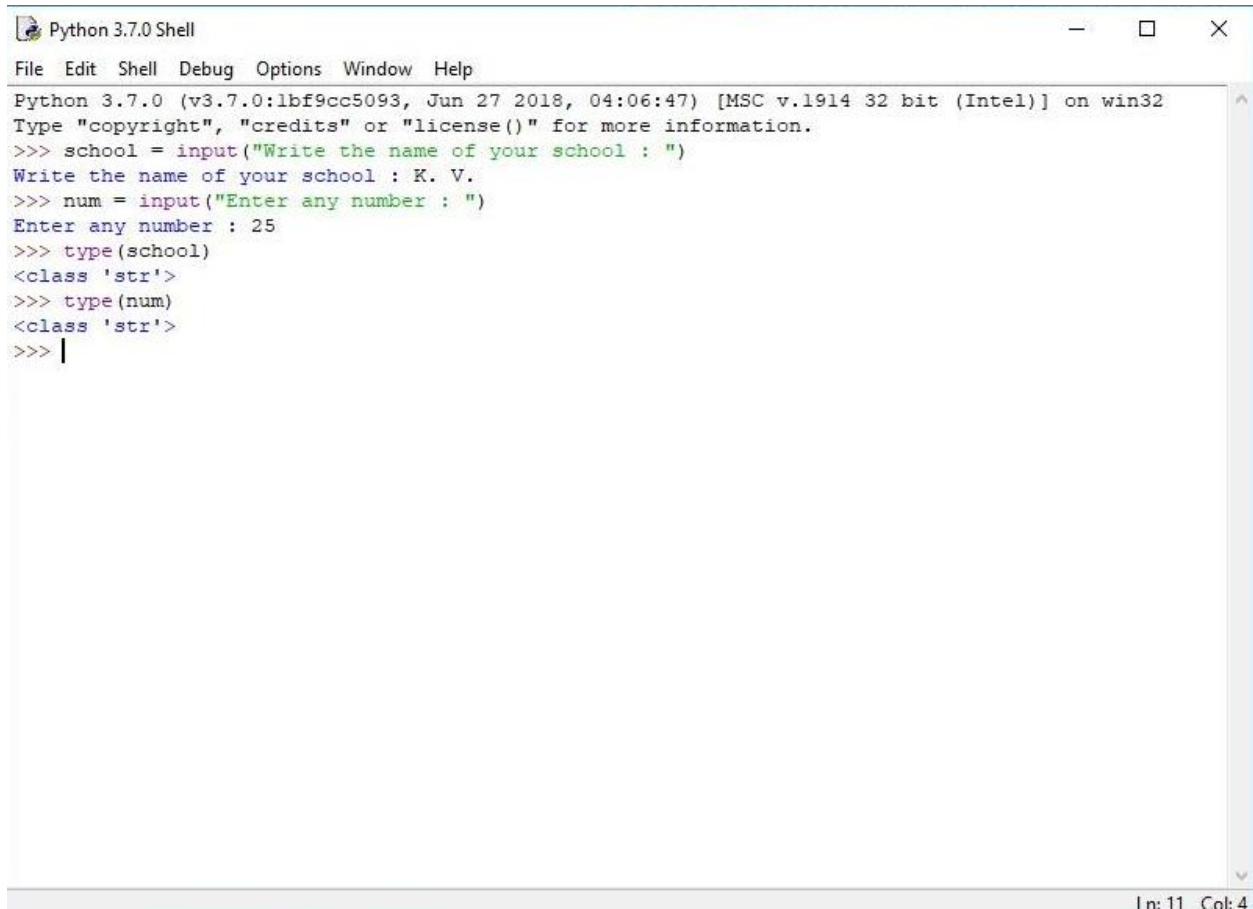
`input()` function is used to get data from the user while working with the script mode.

`type()` function is used to know the data type of an object.

Example :

`type(school)`

`type(num)`



```
Python 3.7.0 Shell
File Edit Shell Debug Options Window Help
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:06:47) [MSC v.1914 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> school = input("Write the name of your school : ")
Write the name of your school : K. V.
>>> num = input("Enter any number : ")
Enter any number : 25
>>> type(school)
<class 'str'>
>>> type(num)
<class 'str'>
>>> |
```

Ln: 11 Col: 4

`input()` enables us to accept an input string from the user without evaluating its value.

`print()` function is used to print the output on the screen.

Example :

```
print("Welcome to ", school)
```

```
Python 3.7.0 Shell
File Edit Shell Debug Options Window Help
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:06:47) [MSC v.1914 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> school = input("Write the name of your school : ")
Write the name of your school : K. V.
>>> num = input("Enter any number : ")
Enter any number : 25
>>> type(school)
<class 'str'>
>>> type(num)
<class 'str'>
>>> print("Welcome To ", school)
Welcome To  K. V.
>>> |
```

Reading Numbers :

Syntax for Input Function with predefined data type for numbers :

Variable = data-type(input("<message>"))

Example :

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

[If the user inputs a decimal number then it will show an error, as 'a' is defined as an integer.]

b = float(input("Enter any number : "))

c = input("Enter any number : ")

type(a)

type(b)

type(c)

```
Python 3.7.0 Shell
File Edit Shell Debug Options Window Help
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:06:47) [MSC v.1914 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> school = input("Write the name of your school : ")
Write the name of your school : K. V.
>>> num = input("Enter any number : ")
Enter any number : 25
>>> type(school)
<class 'str'>
>>> type(num)
<class 'str'>
>>> print("Welcome To ", school)
Welcome To K. V.
>>> a = int(input("Enter any number : "))
Enter any number : 2.5
Traceback (most recent call last):
  File "<pysshell#5>", line 1, in <module>
    a = int(input("Enter any number : "))
ValueError: invalid literal for int() with base 10: '2.5'
>>> a = int(input("Enter any number : "))
Enter any number : 25
>>> b = float(input("Enter any number : "))
Enter any number : 62.5
>>> c = input("Enter any number : ")
Enter any number : 26.6
>>> type(a)
<class 'int'>
>>> type(b)
<class 'float'>
>>> type(c)
<class 'str'>
|>>>
```

Without predefined data type, a variable can be assigned to any value using assignment operator. Data Type will be automatically fixed as per the value.

```
>>> x = 2
>>> y = "Apple"
>>> z = 8.3
>>> print(x)
2
>>> print(y)
Apple
>>> print(z)
8.3
>>> print(type(x))
<class 'int'>
>>> print(type(y))
<class 'str'>
>>> print(type(z))
<class 'float'>
>>>
```

<p><code>+=</code> added and assign back the result to left operand</p>	<p><code>>>>x+=2</code></p>
<p><code>-=</code> subtracted and assign back the result to left operand</p>	<p><code>x-=2</code></p>
<p><code>*=</code> multiplied and assign back the result to left operand</p>	<p><code>x*=2</code></p>
<p><code>/=</code> divided and assign back the result to left operand</p>	<p><code>x/=2</code></p>
<p><code>%=</code> taken modulus using two operands and assign the result to left operand</p>	<p><code>x%=2</code></p>
<p><code>**=</code> performed exponential (power) calculation on operators and assign value to the left operand</p>	<p><code>x**=2</code></p>
<p><code>//=</code> performed floor division on operators and assign value to the left operand</p>	<p><code>x // = 2</code></p>

Assignment

Write a code in Python that accepts two integers from user and print their sum.

```
>>> def fun3():  
    a = int(input("Enter any Integer : "))  
    b = int(input("Enter any Integer : "))  
    c = a + b  
    print("Sum of the Integers is ", c)
```

```
>>> fun3()  
Enter any Integer : -18  
Enter any Integer : 8  
Sum of the Integers is  -10  
>>> |
```

Unsolved :

Q.1.

Create following Variables

- i) "mystring" to contain "Hello"
- ii) "myfloat" to contain "2.5"
- iii) "myint" to contain "10"

Q.2.

If the value of $a = 20$ and $b = 20$, then $a+=b$ will assign _____ to a.

OPERATORS AND EXPRESSIONS

Operators are the constructs which can manipulate the value of operands.

Consider the expression $4 + 5 = 9$. Here, 4 and 5 are called operands and + is called operator.

Types of Operator

Python language supports the following types of operators.

- Arithmetic Operators
- Comparison (Relational) Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

Let us have a look on all the operators one by one.

Python Arithmetic Operators

Assume variable a holds 10 and variable b holds 20, then –

Operator	Description	Example
+ Addition	Adds values on either side of the operator.	$a + b = 30$
- Subtraction	Subtracts right hand operand from left hand operand.	$a - b = -10$

* Multiplication	Multiplies values on either side of the operator	$a * b = 200$
/ Division	Divides left hand operand by right hand operand	$b / a = 2$
% Modulus	Divides left hand operand by right hand operand and returns remainder	$b \% a = 0$
** Exponent	Performs exponential (power) calculation on operators	$a^{**}b = 10$ to the power 20
// Floor Division	The division of operands where the result is the quotient in which the digits after the decimal point are removed. But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity) –	$9 // 2 = 4$ $9.0 // 2.0 = 4.0,$ $-11 // 3 = -4,$ $-11.0 // 3 = -4.0$

Python Comparison Operators

These operators compare the values on either sides of them and decide the relation among them. They are also called Relational operators.

Assume variable a holds 10 and variable b holds 20, then –

Operator	Description	Example
----------	-------------	---------

==	If the values of two operands are equal, then the condition becomes true.	(a == b) is not true.
!=	If values of two operands are not equal, then condition becomes true.	(a != b) is true.
<>	If values of two operands are not equal, then condition becomes true.	(a <> b) is true. This is similar to != operator.
>	If the value of left operand is greater than the value of right operand, then condition becomes true.	(a > b) is not true.
<	If the value of left operand is less than the value of right operand, then condition becomes true.	(a < b) is true.
>=	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.	(a >= b) is not true.
<=	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.	(a <= b) is true.

Python Assignment Operators

Assume variable a holds 10 and variable b holds 20, then –

Operator	Description	Example
=	Assigns values from right side operands to left side operand	c = a + b assigns value of a + b into c
+= Add AND	It adds right operand to the left operand and assign the result to left operand	c += a is equivalent to c = c + a
-= Subtract AND	It subtracts right operand from the left operand and assign the result to left operand	c -= a is equivalent to c = c - a
*= Multiply AND	It multiplies right operand with the left operand and assign the result to left operand	c *= a is equivalent to c = c * a
/= Divide AND	It divides left operand with the right operand and assign the result to left operand	c /= a is equivalent to c = c / a c /= a is equivalent to c = c /

		a
%= Modulus AND	It takes modulus using two operands and assign the result to left operand	c %= a is equivalent to c = c % a
**= Exponent AND	Performs exponential (power) calculation on operators and assign value to the left operand	c **= a is equivalent to c = c ** a
//= Floor Division	It performs floor division on operators and assign value to the left operand	c //= a is equivalent to c = c // a

Python Bitwise Operators

Bitwise operator works on bits and performs bit by bit operation. Assume if a = 60; and b = 13; Now in binary format they will be as follows -

a = 0011 1100

b = 0000 1101

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a = 1100 0011

There are following Bitwise operators supported by Python language

Operator	Description	Example
& Binary AND	Operator copies a bit to the result if it exists in both operands	(a & b) (means 0000 1100)
Binary OR	It copies a bit if it exists in either operand.	(a b) = 61 (means 0011 1101)
^ Binary XOR	It copies the bit if it is set in one operand but not both.	(a ^ b) = 49 (means 0011 0001)
~ Binary Ones Complement	It is unary and has the effect of 'flipping' bits.	(~a) = -61 (means 1100 0011 in 2's compleme nt form due to a signed binary number.
<< Binary Left	The left operands value is moved	a << 2 =

Shift	left by the number of bits specified by the right operand.	240 (means 1111 0000)
>> Binary Right Shift	The left operands value is moved right by the number of bits specified by the right operand.	a >> 2 = 15 (means 0000 1111)

Python Logical Operators

There are following logical operators supported by Python language. Assume variable a holds 10 and variable b holds 20 then

-

Operator	Description	Example
and Logical AND	If both the operands are true then condition becomes true.	(a and b) is false.
or Logical OR	If any of the two operands are non-zero then condition becomes true.	(a or b) is true.
not Logical NOT	Used to reverse the logical state of its operand.	not(a and b) is true.

Python Membership Operators

Python's membership operators test for membership in a sequence, such as strings, lists, or tuples. There are two membership operators as explained below –

Operator	Description	Example
in	Evaluates to true if it finds a variable in the specified sequence and false otherwise.	x in y, here in results in a 1 if x is a member of sequence y.
not in	Evaluates to true if it does not find a variable in the specified sequence and false otherwise.	x not in y, here not in results in a 1 if x is not a member of sequence y.

Python Identity Operators

Identity operators compare the memory locations of two objects. There are two Identity operators explained below –

Operator	Description	Example
is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.	x is y, here is results in 1 if id(x) equals id(y).
is not	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.	x is not y, here is not results in 1 if id(x) is not equal to id(y).

Precedence of operators

- Precedence—when an expression contains two different kinds of operators, which should be applied first is known as operator precedence.

- Associativity—when an expression contains two operators with the same precedence, which should be applied first is known as associativity.

The operators can be listed from high precedence to low precedence as follows:

Operator	Description	Associativity
**	Exponentiation (raise to the power)	Right to Left
+ , -	unary plus and minus	Left to Right
* , / , % , //	Multiply, divide, modulo and floor division	Left to Right
+ , -	Addition and subtraction	Left to Right
< , <= , > , >=	Comparison operators	Left to Right

==, !=	Equality operators	Left to Right
% =, / =, // =, - =, + =, * =	Assignment operators	Right to Left
not and or	Logical operators	Left to Right

Example:

Evaluate the following expression with precedence of operator:

$$X = 2 * 3 ** 2 / 5 + 10 // 3 - 1$$

Ans: $2 * 3 ** 2 / 5 + 10 // 3 - 1$

$$\underline{2 * 9} / 5 + 10 // 3 - 1$$

$$\underline{18} / 5 + 10 // 3 - 1$$

$$3 + \underline{10 // 3} - 1$$

$$\underline{3 + 3} - 1$$

$$\underline{6} - 1$$

Ans = 5

Expressions:

An **expression** is a combination of values, variables, operators, and calls to functions. Expressions need to be evaluated. If you ask Python to **print** an expression, the interpreter **evaluates** the expression and displays the result.

Example1:

```
value1 = eval(input('Please enter a number: '))
```

```
value2 = eval(input('Please enter another number: '))
```



```
sum = value1 + value2
print(value1, '+', value2, '=', sum)
```

The *evaluation of an expression* produces a value, which is why expressions can appear on the right hand side of assignment statements. A value all by itself is a simple expression, and so is a variable. Evaluating a variable gives the value that the variable refers to.

A **statement** is an instruction that the Python interpreter can execute. We have only seen the assignment statement so far. Some other kinds of statements that we'll see shortly are **while** statements, **for** statements, **if** statements, and **import** statements.

If we take a look at this same example in the Python shell, we will see one of the distinct differences between statements and expressions.

```
>>> y = 3.14
>>> x = len("hello")
>>> print(x)
5
>>> print(y)
3.14
>>> y
3.14
```

```
>>>
```

Note that when we enter the assignment statement, $y = 3.14$, only the prompt is returned. There is no value. This is due to the fact that statements, such as the assignment statement, do not return a value. They are simply executed.

On the other hand, the result of executing the assignment statement is the creation of a reference from a variable, y , to a value, 3.14 . When we execute the print function working on y , we see the value that y is referring to. In fact, evaluating y by itself results in the same response.

SAMPLE QUESTIONS AND ANSWERS

1. Write the following expressions using operators used in Python:

(i) $c = \frac{a+b}{2}$

(ii) $x = a^3 + b^3 + c^3$

(iii) $A = \pi r (r + h)^2$

(iv) $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

Answer: 1. $c = (a + b) / (2 * a)$

2. $x = a^{**3} + b^{**3} + c^{**3}$

3. $A = \text{math.pi} * r (r + h) ** 2$ or $A = 3.14 * r (r + h) ** 2$

4. $x = (-b + \text{math.sqrt}(b*b - 4 * a * c)) / (2 * a)$

2. Which is the correct operator for power(x^y)?

a) X^y

b) $X^{}y$**

c) $X^{^}y$

d) None of the mentioned

3. Which one of these is floor division?

a) /

b) //

c) %

d) None of the mentioned

4. What is the order of precedence in python?

i) Parentheses

ii) Exponential

iii) Multiplication

iv) Division

v) Addition

vi) Subtraction

a) i,ii,iii,iv,v,vi

b) ii,i,iii,iv,v,vi

c) ii,i,iv,iii,v,vi

d) i,ii,iii,iv,vi,v

5. What is answer of this expression, $22 \% 3$ is?

a) 7

b) 1

c) 0

d) 5

6. Mathematical operations can be performed on a string. State whether true or false.

a) True

b) False

7. Operators with the same precedence are evaluated in which manner?

a) Left to Right

b) Right to Left

c) Can't say

d) None of the mentioned

8. What is the output of this expression, $3*1**3$?

a) 27

b) 9

c) 3

d) 1

9. Which one of the following have the same precedence?

a) Addition and Subtraction

b) Multiplication and Division

c) Both Addition and Subtraction AND Multiplication and Division

d) None of the mentioned

10. The expression $\text{Int}(x)$ implies that the variable x is converted to integer. State whether true or false.

a) True

b) False

11. Which one of the following have the highest precedence in the expression?

a) Exponential

b) Addition

c) Multiplication

d) Parentheses

12. The value of the expressions $4/(3*(2-1))$ and $4/3*(2-1)$ is the same. State whether true or false.

a) True

b) False

13. The value of the expression:

$4 + 3 \% 5$

a) 4

b) 7

c) 2

d) 0

14. Evaluate the expression given below if A= 16 and B = 15.

$A \% B // A$

a) 0.0

b) 0

c) 1.0

d) 1

15. Which of the following operators has its associativity from right to left?

a) +

b) //

c) %

d) **

16. What is the value of x if:

$x = \text{int}(43.55 + 2 / 2)$

a) 43

b) 44

c) 22

d) 23

17. What is the value of the following expression?

$2 + 4.00, 2 ** 4.0$

a) (6.0, 16.0)

b) (6.00, 16.00)

c) (6, 16)

d) (6.00, 16.0)

18. Which of the following is the truncation division operator?

a) /

b) %

c) //

d) |

19. What are the values of the following expressions:

$2 ** (3 ** 2)$

$(2 ** 3) ** 2$

$2 ** 3 ** 2$

- a) 64, 512, 64
- b) 64, 64, 64
- c) 512, 512, 512
- d) 512, 64, 512**

20. What is the value of the following expression:

`8/4/2, 8/(4/2)`

- a) (1.0, 4.0)**
- b) (1.0, 1.0)
- c) (4.0, 1.0)
- d) (4.0, 4.0)

21. What is the value of the following expression:

`float(22//3+3/3)`

- a) 8
- b) 8.0**
- c) 8.3
- d) 8.33

22. How many types of operators Python has? Give brief idea about them

Python has five types of operators. They are

- **Arithmetic Operators** : This operators are used to do arithmetic operations
- **Comparison Operators** : This operators are used to do compare between two variables of same data-type.
- **Bitwise Operators** : This kind of operators are used to perform bitwise operation between two variable
- **Logical Operators** : This operators performs logical AND, OR, NOT operations among two expressions.
- **Python Assignment Operators** : This operators are used to perform both arithmetic and assignment operations altogether.

23. What is the output of the following code and why?

```
a = 2
b = 3
c = 2

if a == c and b != a or b == c:
    print("if block: executed")
    c = 3

if c == 2:
    print("if block: not executed")
```

The output of the following code will be

if block: executed

This happens because logical AND operator has more precedence than logical OR operator. So **a == c** expression is true and **b != a** is also true. So, the result of logical AND operation is true. As one variable of OR operation is true. So the result of Logical operation is also true. And that why the statements under first if block executed. So the value of variable **c** changes from 2 to 3. And, As the value of C is not true. So the statement under second block doesn't execute.

24. What value is printed when the following statement executes?

```
print(18 / 4)
```

- (A) 4.5
- (B) 5
- (C) 4
- (D) 2

25. What value is printed when the following statement executes?

```
print(18 // 4)
```

- (A) 4.25
- (B) 5
- (C) 4
- (D) 2

26: What value is printed when the following statement executes?

```
print(18 % 4)
```

- (A) 4.25
- (B) 5
- (C) 4
- (D) 2

PYTHON IF...ELIF...ELSE STATEMENTS

An else statement can be combined with an if statement. An else statement contains the block of code that executes if the conditional expression in the if statement resolves to 0 or a FALSE value.

The else statement is an optional statement and there could be at most only one else statement following if.

Syntax

The syntax of the if...else statement is –

if expression:

 statement(s) else:

statement(s)

Python if Statement Flowchart

Example

```
#!/usr/bin/python
```



```

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
1 / 3
else:
    print "2 - Got a false expression value"
    print var2
print "Good bye!"

```

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

```

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

```

The elif Statement

The elif statement allows you to check multiple expressions for TRUE and execute a block of code as soon as one of the conditions evaluates to TRUE.

Similar to the else, the elif statement is optional. However, unlike else, for which there can be at most one statement, there can be an arbitrary number of elif statements following an if.

syntax

```

if expression1:

```

```

    statement(s)
elif

```

```

    expression2: statement(s)
elif expression3:

```

```

    statement(s)
else:

```

```

    statement(s)

```

Core Python does not provide switch or case statements as in other languages, but we can use if..elif...statements to simulate switch case as follows –

Example

```
#!/usr/bin/python
```

```
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
```

```
2 /3
```

```
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!"
```

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

3 - Got a true expression value 100 Good bye!

If the number is positive, we print an appropriate message

```
num = 3 if num > 0:
```

```
    print(num, "is a positive number.") print("This is always printed.")
```

```
num = -1 if num > 0:
```

```
    print(num, "is a positive number.") print("This is also always  
printed.")
```

3 is a positive number This is always printed This is also always printed.

Flowchart of if...elif...else

3 /3

Python Nested if Example

In this program, we input a number # check if the number is positive or # negative or zero and display # an appropriate message # This time we use nested if

```
num = float(input("Enter a number: ")) if num >= 0:
```

```
if num == 0:
```

```
    print("Zero") else:
```

```
        print("Positive number") else:
```

```
print("Negative number")
```

Output Enter a number: 5 Positive number

4 /3

Simple pyramid pattern

```
# Python 3.x code to demonstrate star pattern
# Function to demonstrate printing pattern def pypart(n):
# outer loop to handle number of rows # n in this case for i in
range(0, n):
# inner loop to handle number of columns # values changing acc. to
outer loop for j in range(0, i+1):
# printing stars print("* ",end="")
# ending line after each row print("\r")
# Driver Code n = 5 pypart(n)
```

Output

```
* * * * *
```

```
Printing Triangle # Function to demonstrate printing pattern
triangle def triangle(n):
```

```
# number of spaces k = 2*n - 2
# outer loop to handle number of rows for i in range(0, n):
# inner loop to handle number spaces # values changing acc. to
requirement for j in range(0, k): print(end=" ")
# decrementing k after each loop k = k - 1
```

5 /3

```
# inner loop to handle number of columns # values changing acc. to
outer loop for j in range(0, i+1):
# printing stars print("* ", end="")
# ending line after each row print("\r")
# Driver Code n = 5 triangle(n)
```

Output

```
* * * * *
```

6 /3

FOR LOOP WHILE LOOP AND PROGRAMS BASED ON THAT

Python has two types of loop structure:

- 1.While loop
- 2.For loop

While Loop:-

It iterates till the condition is true:-
It is an entry controlled loop

```
i = 1
while i < 6:
print(i)
i += 1
```

Remember to update the value of i ,otherwise it will continue forever.

The while loop requires relevant variables to be ready, in this example we need to define an indexing variable, i, which we set to 1.

Following is the structure of while loop:-

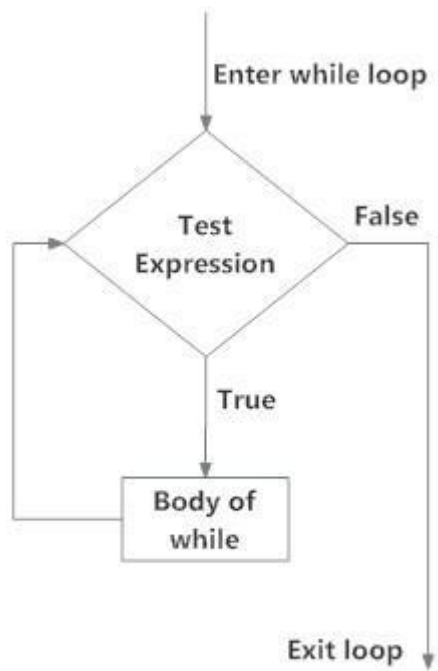


Fig: operation of while loop

The break Statement

With the break statement we can stop the loop even if the while condition is true:

Exit the loop when i is 3

```
i = 1
```

```
while i < 6:  
    print(i)  
    if i == 3:
```

```
    break
    i += 1
```

The continue Statement

With the continue statement we can stop the current iteration, and continue with the next:

Example:-

```
i = 0
while i < 6:
    i += 1
    if i == 3:
        continue
    print(i)
```

FOR LOOP

A for loop is used for iterating over a sequence (that is either a list, a tuple or a string or just a series of numbers).

With for loop we can execute a set of statements, once for each item in a list, tuple, set etc

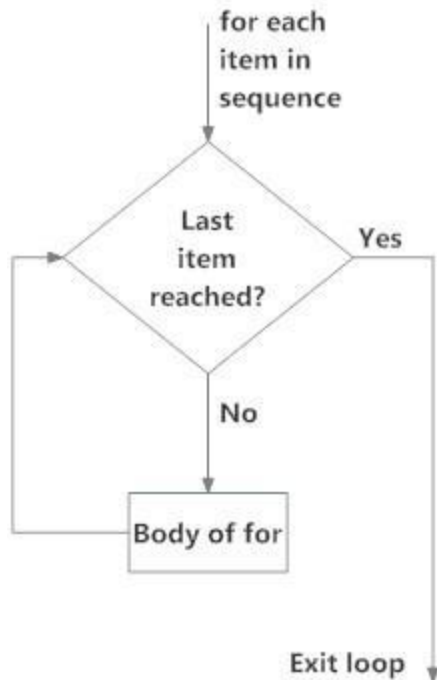


Fig: operation of for loop

Example

Print each fruit in a fruit list:

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    print(x)
```

The for loop does not require an indexing variable to set beforehand, as the for command itself allows for this.

The break Statement

With the break statement we can stop the loop before it has looped through all the items:

Exit the loop when x is “banana”

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    if x == "banana":
```

```
break
print(x)
```

The continue Statement

With the continue statement we can stop the current iteration of the loop, and continue with the next:

Example:-“Do not print banana”

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    if x == "banana":
        continue
    print(x)
```

The range() Function

To loop through a set of code a specified number of times, we can use the range() function,

The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

Example :- range()

```
for x in range(6):
    print(x)
```

Note that range(6) is not the values of 0 to 6, but the values 0 to 5.

The range() function defaults to 0 as a starting value, however it is possible to specify the starting value by adding a parameter:

range(2, 6), which means values from 2 to 6 (but not including 6):

```
for x in range(2, 6):
    print(x)
```

The range() function defaults to increment the sequence by 1, however it is possible to specify the increment value by adding a third parameter: range(2, 30, **3**):

Example:-Using the sequence with 3.

```
for x in range(2, 30, 3):  
    print(x)
```

Example Programs:-

1. Python Program to Calculate the Average of Numbers in a Given List
2. Python Program to Reverse a Given Number
3. Python Program to Take in the Marks of 5 Subjects and Display the Grade
4. Python Program to Read Two Numbers and Print Their Quotient and Remainder
5. Python Program to Accept Three Digits and Print all Possible Combinations from the Digits
6. Python Program to Print Odd Numbers Within a Given Range
7. Python Program to Find the Sum of Digits in a Number
8. Python Program to Find the Smallest Divisor of an Integer
9. Python Program to Count the Number of Digits in a Number
10. Python Program to Check if a Number is a Palindrome
11. Python Program to Read a Number n And Print the Series "1+2+.....+n= "
12. Python Program to Read a Number n and Print the Natural Numbers Summation Pattern

Write a lot of programs: interest calculation, primarily testing, and factorials.

1. Write a program to input principal amount, rate and time and calculate the simple interest and amount after each of the year.

```

p= int(input( "ENTER PRINCIPAL AMOUNT : "))
r= int(input( "ENTER RATE : "))
t= int(input( "ENTER TIME : "))

for i in range (1,t+1):
    interest= (p*r*i)/100
    amount= p+ interest
    print ("AFTER ",i, " YEAR ", " INTEREST = ",interest,
"AMOUNT = ",amount)

```

2. Write a program to input principal amount, rate and time and calculate the compound interest and amount after each of the year.

```

p= int(input( "ENTER PRINCIPAL AMOUNT : "))
r= int(input( "ENTER RATE : "))
t= int(input( "ENTER TIME : "))

for i in range (1,t+1):
    interest= (p*r*1)/100
    p= p+ interest
    print ("AFTER ",i, " YEAR ", " INTEREST = ",interest,
"AMOUNT = ",p)

```

3. Write a program to input a number and print the factorial of the number.

```

num= int(input( "ENTER NUMBER : "))
# check if the number is negative, positive or zero
if num < 0:
    print("Sorry, factorial does not exist for negative numbers")
elif num == 0:

```

```

    print("The factorial of 0 is 1")
else:
    for i in range(1,num + 1):
        factorial = factorial*i
    print("The factorial of",num,"is",factorial)

```

4. Write a program to input a no and check whether prime or not.

```

Num = int(input( "ENTER NUMBER : "))
if num > 1:
    # check for factors
    for i in range(2,num):
        if (num % i) == 0:
            print(num,"is not a prime number")
            break
    else:
        print(num,"is a prime number")

# if input number is less than
# or equal to 1, it is not prime
else:
    print(num," is not a prime number")

```

5. Write a program in python to display all the prime numbers within an interval .

```

lower = int( input("enter lower range ")
upper = int( input("enter upper range ")

print("Prime numbers between",lower,"and",upper,"are:")

for num in range(lower,upper + 1):
    # prime numbers are greater than 1
    if num > 1:

```

```
for i in range(2,num):
    if (num % i) == 0:
        break
else:
    print(num)
```

6. Write a program in python to print the multiplication table of a no.

```
num = 12
```

```
# To take input from the user
num = int(input("Display multiplication table of? "))

# use for loop to iterate 10 times
for i in range(1, 11):
    print(num,'x',i,'=',num*i)
```

7. Write a Program to display the Fibonacci sequence up to n-th term where n is provided by the user.

```
nterms = int(input("How many terms? "))

# first two terms
n1 = 0
n2 = 1
count = 0

# check if the number of terms is valid
if nterms <= 0:
    print("Please enter a positive integer")
elif nterms == 1:
    print("Fibonacci sequence upto",nterms,":")
    print(n1)
else:
```

```

print (n1,n2, end=' ')
print("Fibonacci sequence upto",nterms,":")
while count < nterms:
    print(n1,end=' ')
    nth = n1 + n2
    n1 = n2
    n2 = nth
    count += 1

```

8. Write a program to input a no and check armstrong no or not

```

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

```

```

sum=0

```

```

temp= num

```

```

while temp>0:

```

```

    digit = temp % 10

```

```

    sum += digit**3

```

```

    temp //=10

```

```

if num == sum:

```

```

    print (num, "is Armstrong no")

```

```

else:

```

```

    print (num, "is not Armstrong no")

```

9. Write a Program to check Armstrong numbers in certain interval.

```

# To take input from the user

```

```

lower = int(input("Enter lower range: "))

```

```

upper = int(input("Enter upper range: "))

```

```

for num in range(lower, upper + 1):

```

```

order = len(str(num))
sum = 0

# find the sum of the cube of each digit
temp = num
while temp > 0:
    digit = temp % 10
    sum += digit ** order
    temp //= 10

if num == sum:
    print(num)

```

10. Write a Python program to find the sum of natural numbers up to n where n is provided by user

```

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

if num < 0:
    print("Enter a positive number")
else:
    sum = 0
    # use while loop to iterate un till zero
    while(num > 0):
        sum += num
        num -= 1
    print("The sum is",sum)

```

11. Write Python Program to find the L.C.M. of two inputted number

```

x= int(input("enter first no "))
y= int(input("enter second no "))

if x > y:
    greater = x

```



```
else:
    greater = y

while(True):
    if((greater % x == 0) and (greater % y == 0)):
        lcm = greater
        break
    greater += 1
return lcm
```

12 Write Python Program to find the factors of a number x

```
x= int(input("enter no "))
print("The factors of",x,"are:")

for i in range(1, x + 1):
    if x % i == 0:
        print(i)
```

13. Write python Program to count the number of each vowel in a string

```
vowels = 'aeiou'

ip_str = ip_str.casefold()
count = 0
# count the vowels
for char in ip_str:
    if char in count:
        count[char] += 1

print(count)
```

14. Write Python program to display output-

1

2 2

3 3 3

4 4 4 4

5 5 5 5 5

```
for i in range(1, 5):
    for j in range(i):
        print(i, end=' ')
    print()
```

15. Write the python program to print the following

```
*
* *
* * *
* * * *
* * * * *

n=6
for i in range(1,n):
    for j in range(i):
        print(j, end=' ')
    print()
```

16. Write a Python program to find those numbers which are divisible by 7 and multiple of 5, between 1500 and 2700 (both included).

```
for x in range(1500, 2701):
    if (x%7==0) and (x%5==0):
```

```
print (x)
```

17. Write a Python program to guess a number between 1 to 9.

```
import random
target_num, guess_num = random.randint(1, 10),0
while target_num != guess_num:
    print target_num
    guess_num = int(input('Guess a number between 1 and
10 until you get it right : '))
print('Well guessed!')
```

18. Write a Python program that accepts a word from the user and reverse it.

```
word = input("Input a word to reverse: ")

for char in range(len(word) - 1, -1, -1):
    print(word[char], end="")
print("\n")
```

19 Write a Python program to count the number of even and odd numbers from a series of numbers 5 to 100.

```
count_odd = 0
count_even = 0
for x in range(1,101):
    if not x % 2:
        count_even+=1
    else:
        count_odd+=1
print("Number of even numbers :",count_even)
```

```
print("Number of odd numbers :",count_odd)
```

20. Write a Python program that accepts a string and calculate the number of digits and letters.

```
s = input("Input a string")
d=l=0
for c in s:
    if c.isdigit():
        d=d+1
    elif c.isalpha():
        l=l+1
    else:
        pass
print("Letters", l)
print("Digits", d)
```

21 Write a Python program to find numbers between 100 and 400 (both included) where each digit of a number is an even number.

```
for i in range(100, 401):
    s = str(i)
    if (int(s[0])%2==0) and (int(s[1])%2==0) and (int(s[2])%2==0):
        print( i))
```

TYPES OF ERRORS AND EXCEPTIONS IN PYTHON

Syntax Errors

When you forget a colon at the end of a line, accidentally add one space too many when indenting under an if statement, or forget a parenthesis, you will encounter a syntax error. This means that Python couldn't figure out how to read your program. This is similar to forgetting punctuation in English: for example, this text is difficult to read there is no punctuation there is also no capitalization why is this hard because you have to figure out where each sentence ends you also have to figure out where each sentence begins to some extent it might be ambiguous if there should be a sentence break or not

People can typically figure out what is meant by text with no punctuation, but people are much smarter than computers. If Python doesn't know how to read the program, it will just give up and inform you with an error. For example:

```
defsome_function()
msg = "hello, world!"
print(msg)
returnmsg
File "<ipython-input-3-6bb841ea1423>", line 1
defsome_function()
      ^
```

SyntaxError: invalid syntax

Here, Python tells us that there is a `SyntaxError` on line 1, and even puts a little arrow in the place where there is an issue. In this case the problem is that the function definition is missing a colon at the end.

Actually, the function above has *two* issues with syntax. If we fix the problem with the colon, we see that there is *also* an `IndentationError`, which means that the lines in the function definition do not all have the same indentation:

```
defsome_function():
msg = "hello, world!"
print(msg)
returnmsg
File "<ipython-input-4-ae290e7659cb>", line 4
returnmsg
  ^
```

IndentationError: unexpected indent

Both `SyntaxError` and `IndentationError` indicate a problem with the syntax of your program, but an `IndentationError` is more specific: it *always* means that there is a problem with how your code is indented.

Syntax errors are the most basic type of error. They arise when the Python parser is unable to understand a line of code. Syntax errors

are almost always fatal, i.e. there is almost never a way to successfully execute a piece of code containing syntax errors. Some syntax errors can be caught and handled, like `eval("")`, but these are rare.

In IDLE, it will highlight where the syntax error is. Most syntax errors are typos, incorrect indentation, or incorrect arguments. If you get this error, try looking at your code for any of these.

Logic errors

These are the most difficult type of error to find, because they will give unpredictable results and may crash your program. A lot of different things can happen if you have a logic error. However these are very easy to fix as you can use a debugger, which will run through the program and fix any problems.

Tabs and Spaces

Some indentation errors are harder to spot than others. In particular, mixing spaces and tabs can be difficult to spot because they are both whitespace. In the example below, the first two lines in the body of the function `some_function` are indented with tabs, while the third line — with spaces. If you're working in a Jupyter notebook, be sure to copy and paste this example rather than trying to type it in manually because Jupyter automatically replaces tabs with spaces.

```
defsome_function():  
    msg = "hello, world!"  
    print(msg)  
returnmsg
```

Visually it is impossible to spot the error. Fortunately, Python does not allow you to mix tabs and spaces.

```
File "<ipython-input-5-653b36fbcd41>", line 4  
returnmsg  
    ^
```

TabError: inconsistent use of tabs and spaces in indentation

Variable Name Errors

Another very common type of error is called a `NameError`, and occurs when you try to use a variable that does not exist. For example:

```
print(a)
```

```
-----  
NameErrorTraceback (most recent call last)  
<ipython-input-7-9d7b17ad5387> in <module>()  
----> 1 print(a)
```

`NameError: name 'a' is not defined`

Variable name errors come with some of the most informative error messages, which are usually of the form “name ‘`the_variable_name`’ is not defined”.

Why does this error message occur? That’s a harder question to answer, because it depends on what your code is supposed to do. However, there are a few very common reasons why you might have an undefined variable. The first is that you meant to use a string, but forgot to put quotes around it:

```
print(hello)
```

```
-----  
NameErrorTraceback (most recent call last)  
<ipython-input-8-9553ee03b645> in <module>()  
----> 1 print(hello)
```

`NameError: name 'hello' is not defined`

The second is that you just forgot to create the variable before using it. In the following example, `count` should have been defined (e.g., with `count = 0`) before the for loop:

```
for number in range(10):  
    count = count + number  
    print("The count is:", count)
```

```
-----  
NameErrorTraceback (most recent call last)  
<ipython-input-9-dd6a12d7ca5c> in <module>()
```

```
1 for number in range(10):
----> 2     count = count + number
      3 print("The count is:", count)
```

NameError: name 'count' is not defined

Finally, the third possibility is that you made a typo when you were writing your code. Let's say we fixed the error above by adding the line `Count = 0` before the for loop. Frustratingly, this actually does not fix the error. Remember that variables are case-sensitive, so the variable `count` is different from `Count`. We still get the same error, because we still have not defined `count`:

```
Count = 0
for number in range(10):
count = count + number
print("The count is:", count)
```

```
-----
NameErrorTraceback (most recent call last)
<ipython-input-10-d77d40059aea> in <module>()
      1 Count = 0
      2 for number in range(10):
----> 3     count = count + number
      4 print("The count is:", count)
```

NameError: name 'count' is not defined

Index Errors

Next up are errors having to do with containers (like lists and strings) and the items within them. If you try to access an item in a list or a string that does not exist, then you will get an error. This makes sense: if you asked someone what day they would like to get coffee, and they answered “aturday”, you might be a bit annoyed. Python gets similarly annoyed if you try to ask it for an item that doesn't exist:

```
letters = ['a', 'b', 'c']
print("Letter #1 is", letters[0])
print("Letter #2 is", letters[1])
```



```
print("Letter #3 is", letters[2])
print("Letter #4 is", letters[3])
Letter #1 is a
Letter #2 is b
Letter #3 is c
```

```
-----
IndexErrorTraceback (most recent call last)
<ipython-input-11-d817f55b7d6c> in <module>()
      3 print("Letter #2 is", letters[1])
      4 print("Letter #3 is", letters[2])
----> 5 print("Letter #4 is", letters[3])
```

IndexError: list index out of range

Here, Python is telling us that there is an IndexError in our code, meaning we tried to access a list index that did not exist.

File Errors

The last type of error we'll cover today are those associated with reading and writing files: FileNotFoundError. If you try to read a file that does not exist, you will receive a FileNotFoundError telling you so. If you attempt to write to a file that was opened read-only, Python 3 returns an UnsupportedOperationError. More generally, problems with input and output manifest as IOError or OSError, depending on the version of Python you use.

```
file_handle = open('myfile.txt', 'r')
```

```
-----
FileNotFoundErrorTraceback (most recent call last)
<ipython-input-14-f6e1ac4aee96> in <module>()
----> 1 file_handle = open('myfile.txt', 'r')
```

FileNotFoundError: [Errno 2] No such file or directory: 'myfile.txt'

Exception Handling in Python – Basic Idea

Exception Handling in Python involves the use of try and except clauses in the following forms wherein the code that may generate an exception is written in the try block and the code for handling exception when the exception is raised, is written in except block.

See below:

try:

```
#write here the code that may generate an exception
```

except:

```
#write code here about what to do the exception has occurred
```

For instance consider the following code:

```
try:
    print("result of 10/5=", (10/5))    The code that may raise
an exception
    print("result of 10/0=", (10/0))  (e.g. a division may have
error by
except:                                zero) is written in try
block.
    print("Divide by Zero Error! Denominator must not be
zero!")
```

The output produced by above code is as shown below:

```
result of 10/5=2
result of 10/0=Divide by Zero Error! Denominator must not be zero!
```

This is when the exception is raised

See, the exception (10/0) raised exception which is then handled by except block

See, now the output produced does not show the scary red-coloured standard error message. It is now what you defined under the exception block.

Python comes with many predefined exceptions, also known as built-in exception. Some common built-in exceptions in Python are listed in the following table.

Some Built – in Exceptions

Exception Name	Description
EOFError	Raised when one of the built-in functions (input()) hits an end-of-file condition (EOF) without reading any data.
IOError	Raised when an I/O operation (such as a print() , the built-in open() function or a method of a file object) fails for an I/O-related reason, e.g., “file not found” or “disk full”.
NameError	Raised when an identifier name is not found.
IndexError	Raised when a sequence subscript or index is out of range, e.g., from a string of length 4 if you try to read a value of index like 4 or more i.e., string[4], string[5], string[-5] etc. will raise exception as legal indexes for a string of length 4 are 0, 1, 2, 3 and -1, -2, -3, -4 only.
ImportError	Raised when an import statement fails to find the module definition or when a from import fails to find a name that is to be imported.
TypeError	Raised when an operation or function is applied to an object of inappropriate type, e.g., if you try to compute a square-root of a string value.
ValueError	Raised when a built-in operation or function receives an argument with inappropriate value e.g., int(“z10”) will raise ValueError.
OverflowError	Raised when the second argument of a division or modulo operation is zero.
KeyError	Raised when the result of an arithmetic operation is too large to be represented.
	Raised when a mapping (dictionary) key is not found in the set of existing keys.

Exceptions

Exceptions arise when the python parser knows what to do with a piece of code but is unable to perform the action. An example would be trying to access the internet with python without an internet connection; the python interpreter knows what to do with that command but is unable to perform it.

Dealing with exceptions

Unlike syntax errors, exceptions are not always fatal. Exceptions can be handled with the use of a try statement.

Consider the following code to display the HTML of the website 'example.com'. When the execution of the program reaches the try statement it will attempt to perform the indented code following, if for some reason there is an error (the computer is not connected to the internet or something) the python interpreter will jump to the indented code below the 'except:' command.

```
import urllib2
url='http://www.example.com'
try:
    req=urllib2.Request(url)
    response=urllib2.urlopen(req)
    the_page=response.read()
    print the_page
except:
    print "We have a problem."
```

Another way to handle an error is to except a specific error.

```
try:
    age=int(raw_input("Enter your age: "))
    print "You must be {0} years old.".format(age)
except ValueError:
    print "Your age must be numeric."
```

If the user enters a numeric value as his/her age, the output should look like this:

```
Enter your age: 5
Your age must be 5 years old.
```

However, if the user enters a non-numeric value as his/her age, a `ValueError` is thrown when trying to execute the `int()` method on a non-numeric string, and the code under the `except` clause is executed:

```
Enter your age: five
Your age must be numeric.
```

You can also use a `try` block with a `while` loop to validate input:

```
valid=False
while valid==False:
    try:
        age=int(raw_input("Enter your age: "))
        valid=True# This statement will only execute if the above statement
        executes without error.
    print"You must be {0} years old.".format(age)
    except ValueError:
        print"Your age must be numeric."
```

The program will prompt you for your age until you enter a valid age:

```
Enter your age: five
Your age must be numeric.
Enter your age: abc10
Your age must be numeric.
Enter your age: 15
You must be 15 years old.
```

In certain other cases, it might be necessary to get more information about the exception and deal with it appropriately. In such situations the `except as` construct can be used.

```
f=raw_input("enter the name of the file:")
l=raw_input("enter the name of the link:")
try:
    os.symlink(f,l)
except OSError as e:
    print"an error occurred linking %s to %s: %s\n error no
    %d"%(f,l,e.args[1],e.args[0])
enter the name of the file:file1.txt
enter the name of the link:AlreadyExists.txt
an error occurred linking file1.txt to AlreadyExists.txt: File exists
```

error no 17

HOW TO DEBUG A PROGRAM

Debugging involves correction of code so that the cause of errors is removed. But how do you know what are the errors? Well, the **compile time errors** (syntax errors and semantic errors) are right during compilation and for **logical errors**, you perform **testing**. Testing is done to verify correct behavior i.e., with some sample values (test cases) whose output is already known, the code is tested – if it produces the anticipated output, code is correct and if it does not, there is some error. Once you know the errors, you can debug your program.

Let us now learn some useful debugging techniques.

Debugging Techniques

Debugging a program is a skill. There are many traditional debugging techniques that you can allow to debug your code. These are:

1. Carefully spot the origin of error.
2. Print variables' intermediate values.
3. Code tracing and stepping.

Carefully spot the Origin of Error

When you run your code. Python interpreter will run a line of code if it is free from any syntax and semantic error, and will list the error if the running line of code has some error, e.g., Consider the following code that inputs a number and then prints its powers from 0 to 3

1. `a = int(input("Enter a number "))`

2. `j = "a"`
3. `for i in range(4):`
4. `print(j**i)`

Upon running the above code, Python reports

file *E:/tem.py*, line4, in <module> See the interpreter reported error in line4

`print(j**i)` But the real error does not be the line4

Type Error : unsupported operand type(s) for ** or pow(): *str' and 'int'

But the statement

`print(j**i)`

is syntactically and semantically correct. Then how to figure out the real origin of the error.

When Python gives you a line number for the error, then:

- (i) It means the error has manifested in this line.
- (ii) Its origin may be in this line or in the lines above it.

So, start looking backwards from the line of error. Carefully reading the statements and you will find the real cause of the error, e.g. for the above error, the real cause of error is line2, where rather than assigning the variable a, we assigned string "a" to variable j.

1. `a = int(input("Enter a number "))`
2. `j = "a"` ← Error occurred because j was mistakenly
3. `for i in range(4):`

4. `print(j**i)` assigned string "a" in place of variable a

So we must correct line 2 as:

```
j = a
```

And now, the code run perfectly fine and gives output as:

```
1
5
25
125
```

2. Print variables' Intermediate values

Depending upon our algorithms, the variables' values change during the execution. Sometimes we get incorrect output but you cannot figure out what is causing it. In that case, it is a good to add many `print()` statements (temporarily) to inspect values of variables after each step e.g.,

Consider the following code that is trying to print some Fibonacci terms:

```
a = 0
b = 1
print(a)
print(b)
for i in range (5):
    c = a+b
    print(c, end = " ")
    b = c
    a = b
```


But it starts producing incorrect terms after printing some correct terms:

```
0
1
1 2 4 8 16
```

Now to debug this code, it is a good idea to temporarily add print() statements with some hint string identifying variable, to check changing values of variables a, b and c, as this:

```
for i in range(5):
    c = a + b
    print("c= ", c)
    a = b
    b = c
    print("a= ", a, end = " ")
    print("b= ", b)
```

The print() statements added to check on intermediate values of variables a and b

Now upon running the code, you can carefully look at the output produced to figure out the error:

```
0
1
c = 1
a= 1 b= 1
c= 2
a= 2 b= 2
c= 4
a= 4 b= 4
```

As per Fibonacci logic, a should have been 1 but it is 2, which means some problem with the assignment: either the assignment statement is incorrect or the order of the

```
c= 8          assignment statements is incorrect
a= 8  b= 8
c= 16
a= 16 b= 16
```

Now carefully look at the code. You will find out that the order of the assignment statements is incorrect: variable b first loses its values before assigning it to variable a, i.e.,:

```
a = b
b = c
```

So correct this and the code now becomes:

```
a = 0
b = 1
print(a)
print(b)
for i in range(5) :
    c = a+b
    print( c )
    a = b
    b = c
```

← This change of order of statements will correct the problem and print the correct Fibonacci terms

3 Code Tracing

Another useful technique is code tracing. Code tracking means executing code one line at a time and watching its impact on variables. Code tracing is often done with built-in debugging tools or debuggers.

USING DEBUGGER TOOL

A **Debugging tool** or debugger is a specialized computer program/software that can be used to test and debug programs written in a specific programming language. A **Debugger** can be a separate program or integrated with IDEs (Integrated Development Environment). Python also provides a separate debugger program called **pdb**. In this section, we shall learn to work with both these types of debuggers. First, we shall learn to work with integrated debugger of **Spyder IDE** of Python and then we shall learn to use **pdb** – separate Python debugger program.

Working with Integrated Debugger Tool of Spyder IDE

Spyder IDE provides a graphical front end to Python's debugger **pdb**, thus makes it easier to use. This is called **winpdb**. Let us find learn to use this GUI debugger of Spyder IDE and later we shall learn to work with Python's command based debugger **pdb**.

To debug your code through Spyder IDE's interactive debugger, follow the steps:

Before you start with debugging process, make sure that Variable Explorer pane is visible. (see on next page)

If, however, this pane is not visible, you can open it using command:

View menus -> Panes -> Variable explorer

Or by pressing shortcut key **Ctrl+Shift+V**.

Q. What is code tracing?

A. Code tracing refers to running the code line by line (interactively) and to show its execution impact on memory variables.

Working with Python Debugger – pdb

Python comes with debugger module – **pdb**. In order to use it for debugging your program interactively, you need to do the following:

(i) Add first line of code to your program as given below:

import pdb

(ii) Add following line, just above the code line from where you want to start tracing of code.

pdb.set_trace()

(iii) Once you have done this and run your program, Python debugger will run the code and show you the following prompt in console window:

ipdb> ←

Here, give pdb commands

You can then give following pdb commands on **ipdb >** prompt and press Enter Key:

Basic pdb Commands	
p	print variable value
n	next
s	step inside a function
c	continue
<Enter>	repeat previous command
l	print nearby code
q	quit
h	help
h command	info about a command
b 45	set break point on line 45
b	list current break points
cl	clear all break points
cl 42	clear break point #42

Let us practically work on pdb to understand it.

Firstly add `import pdb` and `pdb.set_trace()` to the code:

```
import pdb
a = 0
b = 1
print(a)
print(b)
for I in range(5):
    pdb.set_trace( ) ← We want to start code tracing
from here,
    c = a+b
    pdb.set_trace( ) here.
    print(c, end = " ")
    b = c
    a = b
```

thus we would

Q. What is debugging?

A. Debugging refers to the process of locating the place of error, cause of error, and correcting the code accordingly.

Q. What is debugger?

A. Debugger is a tool that lets you trace and execute code line by line.

Q. What is exception?

A. Exception in general refers to some contradictory or unusual situation which can be encountered unexpectedly while executing a program.

Q. Why is Exception Handling required?

A. the exception handling is ideal for processing exceptional situations in a controlled way so that program ends gracefully rather than abrupt crashing of the program.

Q. What is the need for debugger tool?

A. Debugger tools are very useful especially if the code is or the error is not very clear, it becomes very difficult to manually figure out the origin and cause of problem. Debugger tools here prove very handy and useful. They show us the line by line execution and its result on variables interactively and help a programmer get to the root of the problem.

List

A list is a collection which is ordered and changeable. In Python lists are written with square brackets. A List is just like the arrays declared in other languages. But the most powerful thing is that list need not be always homogeneous. A single list can contain strings, integers, as well as objects. Lists can also be used for implementing stacks and queues. Lists are mutable, i.e., they can be altered once declared.

1. Declaring a List :

```
Firstlist = ["France", "Belgium", "England"]
```

Indexing in a list begins from 0

```
#Printing all content of list
```

```
print(Firstlist)
```

```
#declaring few more lists:
```

```
list1 = ['physics', 'chemistry', 1997, 2000];  
list2 = [1, 2, 3, 4, 5 ];  
list3 = ["a", "b", "c", "d"]
```

2. To access values in lists, use the square brackets along with index whose value is to be displayed .

e.g **print list1[0]**

when above statement will execute , **physics will be displayed**

3. Changing Content of a list

```
Firstlist = ["France", "Belgium", "England"]  
Firstlist[2]="Russia"
```

This statement will replace England by Russia

3. An element or more can be added at run time to list using append method

e.g :

```
thislist = ["apple", "banana", "cherry"]  
thislist.append("damson")
```

An element can be removed from the list using remove method

```
thislist = ["apple", "banana", "cherry"]
```

Built-in Functions

1. **Append():** Add an element to the end of the list
2. **Extend():** Add all elements of a list to the another list
3. **Insert():** Insert an item at the defined index
4. **Remove():** Removes an item from the list
5. **Pop():** Removes and returns an element at the given index
6. **Clear():** Removes all items from the list
7. **Index():** Returns the index of the first matched item
8. **Count():** Returns the count of number of items passed as an argument
9. **Sort():** Sort items in a list in ascending order

10.**Reverse():** Reverse the order of items in the list

11.**Copy():** Returns a shallow copy of the list

Few examples:

- 1) Finding maximum and minimum in a list
(using built in function)

```
# declaring list
```

```
a = [3, 10, 1, 3, 4, 5]
```

```
# inbuilt function to find the position of minimum
```

```
minpos = a.index(min(a))
```

```
# inbuilt function to find the position of maximum
```

```
maxpos = a.index(max(a))
```

```
# printing the position
```

```
print ("lowest element is ", a[minpos])
```

```
print ("Highest element is ", a[maxpos])
```

- 2) Finding maximum and minimum in a list

Using loop iteration

```
list1 = [12, 45, 2, 41, 31, 10, 8, 6, 4]
```

```
largest = list1[0]
```

```
lowest = list1[0]
```

```
for item in list1:
```

```
    if item > largest:
```

```
        largest = item
```



```
elif item < lowest:  
    lowest = item
```

```
print("Largest element is:", largest)  
print("Smallest element is:", lowest)
```

3) Implementing Linear search in Python (Using list)

```
items = [5, 7, 10, 12, 15]  
print("list of items is", items)  
  
x = int(input("enter item to search:"))  
  
i = flag = 0  
  
while i < len(items):  
    if items[i] == x:  
        flag = 1  
        break  
  
    i = i + 1  
  
if flag == 1:  
    print("item found at position:", i + 1)  
else:  
    print("item not found")
```

Tuples in Python

- Introduction
- Creating and Accessing Tuples
- Tuple Operations
- Tuple Functions and Methods
- Questions on Tuples

INTRODUCTION

A Tuple is a collection of Python objects separated by commas. In some ways a tuple is similar to a list in terms of indexing, nested objects and repetition but a tuple is immutable unlike lists which are mutable. By immutable we mean that we cannot change the elements of a tuple in place. In fact Python will create a fresh tuple when we make changes to an element of a tuple.

A tuple is a standard data type of Python that can store a sequence of values belonging to any type. The tuples are depicted through parenthesis.

```
() # tuple with no member
(-2,-1,0,1,2,3) #tuple of integers
(1, 2.5, 3, 4, 5.5) # tuple of numbers( integers and floating numbers)
('a',1,',2.6,3,'xyz') #tuple of mixed value types
```

CREATING AND ACCESSING TUPLES

```
1. # An empty tuple
emptytuple = ()
print (emptytuple)
```

It will generate an empty tuple and the name of the tuple is emptytuple.

2. Single Element Tuple

To construct a tuple with one element just add a comma after the single element :

```
Singtup = (10,)
```

Singtup is an example of a tuple with one element.

3. Long Tuple

If a tuple contains several elements it can be split into several lines

```
T=( 2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,  
44,46,48,50,52,54 )
```

4. Nested Tuple

If a Tuple contains an element which is a tuple itself then it is called nested tuple:

```
T=(1,2,(3,4))
```

The tuple contains 3 elements 1, 2 and (3,4) which itself is a tuple.

Advantages of Tuple over List

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

However, there are certain advantages of implementing a tuple over a list.

- We generally use tuple for heterogeneous (different) datatypes and list for homogeneous (similar) datatypes.
- Since tuples 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.
- If you have data that doesn't change, implementing it as tuple will guarantee that it remains write-protected

Accessing Tuples

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 than (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 tuples are accessed using nested indexing, as shown in the example below.

```
my_tuple = ('O','r','a','n','g','e')
print(my_tuple[0])
print(my_tuple[5])
# nested tuple

n_tuple = ("range", [2, 4, 6], (1, 2, 3))
print(n_tuple[0][3])
print(n_tuple[1][1])
```

When we run the program, the output will be:

```
O
e
g
4
```

2. 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 = ('f','o','r','m','a','t')
print(my_tuple[-1])
# Output: 't'
print(my_tuple[-6])
# Output : 'f'
```

3. 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','t')
```

```
# elements 2nd to 4th
```

```
# Output: ('r', 'o', 'g')
```

```
print(my_tuple[1:4])
```

```
# elements beginning to 2nd
```

```
# Output: ('p', 'r')
```

```
print(my_tuple[:2])
```

```
# elements 8th to end
```

```
# Output: ('i', 't')
```

```
print(my_tuple[7:])
```

```
# elements beginning to end
```

```
# Output: ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 't')
```

```
print(my_tuple[:])
```

Modifying Tuples

Since tuples are immutable, so it cannot be modified directly. However, modification is possible by one method illustrated below:

```
T=(10,20,30,40)
```

a. First unpack the tuple:

```
a,b,c,d =T
```

b. Redefine or change desired variable say b

```
b=22
```

c. Now repack the tuple with changed value

```
T= (a,b,c,d)
```

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](#).

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

```
del my_tuple[3] # (Error )
```

```
# can delete entire tuple
```

```
# NameError: name 'my_tuple' is not defined
```

```
del my_tuple
```

```
my_tuple
```

TUPLE FUNCTIONS and METHODS

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

Built-in Functions with Tuple

Function	Description
----------	-------------

<u>len()</u>	Return the length (the number of items) in the tuple.
<u>max()</u>	Return the largest item in the tuple.
<u>min()</u>	Return the smallest item in the tuple
<u>tuple()</u>	Convert an iterable (list, string, set, dictionary) to a tuple.

Linear Search

In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data structure.

```

1. # list of tuples
2. tups = [("a", 1), ("b", 2), ("a", 1), ("c", 3)]
3.
4. # create an index counter to avoid problems with identical
   values
5. c = 0
6.
7. # loop through the list
8. for t in tups:
9.     ...if "a" in t:
10.         ...c+=1
11.if c>0
12.    print("present")
13.else
14.    print("Not present")

```

Python Tuple max() Method

Description

The method **max()** returns the elements from the tuple with maximum value.

Syntax

Following is the syntax for **max()** method –

```
max(tuple)
```

Parameters

- **tuple** – This is a tuple from which max valued element to be returned.

Return Value

This method returns the elements from the tuple with maximum value.

Example

The following example shows the usage of max() method.

```
tuple1, tuple2 = (123, 'xyz', 'zara', 'abc'), (456, 700, 200)
print "Max value element : ", max(tuple1)
print "Max value element : ", max(tuple2)
```

When we run above program, it produces following result –

```
Max value element : zara
Max value element : 700
```

Questions

1. Which of the following is a Python tuple?
 - a) [1, 2, 3].
 - b) (1, 2, 3)
 - c) {1, 2, 3}
 - d) {}
2. Suppose t = (1, 2, 4, 3), which of the following is incorrect?
 - a) print(t[3])
 - b) t[3] = 45
 - c) print(max(t))
 - d) print(len(t))
3. What will be the output?

1. >>>t=(1,2,4,3)
2. >>>t[1:3]

- a) (1, 2)
- b) (1, 2, 4)
- c) (2, 4)
- d) (2, 4, 3)

4. What will be the output?

- 1. `>>>t=(1,2,4,3)`
- 2. `>>>t[1:-1]`

- a) (1, 2)
- b) (1, 2, 4)
- c) (2, 4)
- d) (2, 4, 3)

5. What will be the output?

- `>>>t = (1, 2, 4, 3, 8, 9)`
- `>>[t[i] for i in range(0, len(t), 2)]`

- a) [2, 3, 9].
- b) [1, 2, 4, 3, 8, 9].
- c) [1, 4, 8].
- d) (1, 4, 8)

6. What will be the output?

- 1. `d = {"john":40, "peter":45}`
- 2. `d["john"]`

- a) 40
- b) 45
- c) "john"
- d) "peter"

7. What will be the output?

- 1. `>>>t = (1, 2)`
- 2. `>>>2 * t`

- a) (1, 2, 1, 2)
- b) [1, 2, 1, 2].

- c) (1, 1, 2, 2)
- d) [1, 1, 2, 2].

8. What will be the output?

```
1. >>>t1 = (1, 2, 4, 3)
2. >>>t2 = (1, 2, 3, 4)
3. >>>t1 < t2
```

- a) True
- b) False
- c) Error
- d) None

9. What will be the output?

```
1. >>>my_tuple = (1, 2, 3, 4)
2. >>>my_tuple.append( (5, 6, 7) )
3. >>>print len(my_tuple)
```

- a) 1
- b) 2
- c) 5
- d) Error

10. 0. What will be the output?

```
1. numberGames = {}
2. numberGames[(1,2,4)] = 8
3. numberGames[(4,2,1)] = 10
4. numberGames[(1,2)] = 12
5. sum = 0
6. for k in numberGames:
7.     sum += numberGames[k]
8. print len(numberGames) + sum
```

- a) 30
- b) 24

- c) 33
- d) 12

Answers

1	2	3	4	5	6	7	8	9	10
b	b	c	c	c	a	a	b	d	c

DICTIONARY

A dictionary in Python is a collection of unordered values accessed by key rather than by index. The keys have to be hashable: integers, floating point numbers, strings, tuples, and frozensets are hashable, while lists, dictionaries, and sets other than frozensets are not. Dictionaries were available as early as in Python 1.4.

Overview[

Dictionaries in Python at a glance:

```
dict1 = {} # Create an empty dictionary
dict2 = dict() # Create an empty dictionary 2
dict2 = {"r": 34, "i": 56} # Initialize to non-empty value
dict3 = dict([("r", 34), ("i", 56)]) # Init from a list of tuples
dict4 = dict(r=34, i=56) # Initialize to non-empty value 3
dict1["temperature"] = 32 # Assign value to a key
if "temperature" in dict1: # Membership test of a key AKA key
exists
    del dict1["temperature"] # Delete AKA remove
equalbyvalue = dict2 == dict3
itemcount2 = len(dict2) # Length AKA size AKA item count
isempty2 = len(dict2) == 0 # Emptiness test
for key in dict2: # Iterate via keys
```

```

print key, dict2[key]      # Print key and the associated value
dict2[key] += 10           # Modify-access to the key-value pair
for key in sorted(dict2): # Iterate via keys in sorted order of the
keys
    print key, dict2[key]  # Print key and the associated value
for value in dict2.values(): # Iterate via values
    print value
for key, value in dict2.items(): # Iterate via pairs
    print key, value
dict5 = {} # {x: dict2[x] + 1 for x in dict2 } # Dictionary comprehension
in Python 2.7 or later
dict6 = dict2.copy()      # A shallow copy
dict6.update({"i": 60, "j": 30}) # Add or overwrite; a bit like list's
extend
dict7 = dict2.copy()
dict7.clear()            # Clear AKA empty AKA erase
sixty = dict6.pop("i")   # Remove key i, returning its value
print dict1, dict2, dict3, dict4, dict5, dict6, dict7, equalbyvalue,
itemcount2, sixty

```

Dictionary notation

Dictionaries may be created directly or converted from sequences.
Dictionaries are enclosed in curly braces, {}

```

>>> d = {'city':'Paris', 'age':38, (102,1650,1601):'A matrix coordinate'}
>>> seq = [('city','Paris'), ('age', 38), ((102,1650,1601),'A matrix
coordinate')]
>>> d
{'city': 'Paris', 'age': 38, (102, 1650, 1601): 'A matrix coordinate'}
>>> dict(seq)
{'city': 'Paris', 'age': 38, (102, 1650, 1601): 'A matrix coordinate'}

```

```
>>> d == dict(seq)
True
```

Also, dictionaries can be easily created by zipping two sequences.

```
>>> seq1 = ('a','b','c','d')
>>> seq2 = [1,2,3,4]
>>> d = dict(zip(seq1,seq2))
>>> d
```

```
{'a': 1, 'c': 3, 'b': 2, 'd': 4}
```

Accessing Values in Dictionary

To access dictionary elements, you can use the familiar square brackets along with the key to obtain its value. Following is a simple example –

```
#!/usr/bin/python
```

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
print "dict['Name']: ", dict['Name']
print "dict['Age']: ", dict['Age']
```

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

```
dict['Name']: Zara
```

```
dict['Age']: 7
```

If we attempt to access a data item with a key, which is not part of the dictionary, we get an error as follows –

```
#!/usr/bin/python
```

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
print "dict['Alice']: ", dict['Alice']
```

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

```
dict['Alice']:
Traceback (most recent call last):
  File "test.py", line 4, in <module>
    print "dict['Alice']: ", dict['Alice'];
KeyError: 'Alice'
```

Updating Dictionary

You can update a dictionary by adding a new entry or a key-value pair, modifying an existing entry, or deleting an existing entry as shown below in the simple example –

```
#!/usr/bin/python
```

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
dict['Age'] = 8; # update existing entry
dict['School'] = "DPS School"; # Add new entry

print "dict['Age']: ", dict['Age']
print "dict['School']: ", dict['School']
```

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

```
dict['Age']: 8
dict['School']: DPS School
```

Operations on Dictionaries[edit]

The operations on dictionaries are somewhat unique. Slicing is not supported, since the items have no intrinsic order.

```
>>> d = {'a':1, 'b':2, 'cat': 'Fluffers'}
```

```
>>> d.keys()
['a', 'b', 'cat']
>>> d.values()
[1, 2, 'Fluffers']
>>> d['a']
1
>>> d['cat'] = 'Mr. Whiskers'
>>> d['cat']
'Mr. Whiskers'
>>> 'cat' in d
True
>>> 'dog' in d
False
```

Combining two Dictionaries

You can combine two dictionaries by using the update method of the primary dictionary. Note that the update method will merge existing elements if they conflict.

```
>>> d = {'apples': 1, 'oranges': 3, 'pears': 2}
>>> ud = {'pears': 4, 'grapes': 5, 'lemons': 6}
>>> d.update(ud)
>>> d
{'grapes': 5, 'pears': 4, 'lemons': 6, 'apples': 1, 'oranges': 3}
>>>
```

Deleting from dictionary

```
del dictionaryName[membername]
```

Exercises

Write a program that:

1. Asks the user for a string, then creates the following dictionary. The values are the letters in the string, with the corresponding key being the place in the string.
2. Replaces the entry whose key is the integer 3, with the value "Pie".
3. Asks the user for a string of digits, then prints out the values corresponding to those digits.

COUNTING THE FREQUENCY OF ELEMENTS IN A LIST USING DICTIONARY

 *test.py - C:\Users\Office\Desktop\test.py (3.6.3)*

File Edit Format Run Options Window Help

```
#counting the frequency of elements in
# a list using a dictionary
my_list = [5,5,5,5,2,2,2,3,3,32,6,6,7,8,9]
print("Original List : ",my_list)
my_set = set(my_list)
d=dict.fromkeys(my_set, 0)
print(d)
for n in my_list:
    d[n]=d[n]+1
print(d)
```


Bubble Sort

The basic idea of bubble sort is to compare two adjoining values and exchange them if they are not in proper order. In every pass the heaviest element settles as its appropriate position in the right, next time we need not compare that element i.e after 1st pass we have 1 less number of comparisons in the next pass, we can have 2 less number of comparisons in the next pass and so on and after every pass, the largest value will be at the appropriate position of the array . Like a bubble, it has risen to the top

The complete bubble sort is shown below in a more condensed format.

First Pass

6	3	5	8	2
3	6	5	8	2
3	5	6	8	2
3	5	6	8	2
3	5	6	2	8

Second Pass

3	5	6	2	8
3	5	6	2	8
3	5	6	2	8
3	5	2	6	8

Third Pass

3	5	2	6	8
3	5	2	6	8
3	2	5	6	8

Fourth Pass

3	2	5	6	8
2	3	5	6	8

Program Code for Bubble Sort:

"""Performs a bubble sort on a list of numbers. Returns a sorted list."""

```
def bubbleSort(nlist):  
    for passnum in range(len(nlist)-1,0,-1):  
        for i in range(passnum):  
            if nlist[i]>nlist[i+1]:  
                temp = nlist[i]  
                nlist[i] = nlist[i+1]  
                nlist[i+1] = temp  
  
nlist = [14,46,43,27,57,41,45,21,70]  
bubbleSort(nlist)  
print(nlist)
```

Calculating Number of operations on Bubble Sort:

Number of operations is an important aspect of any algorithms/programs as it specifies the efficiency of the program. Less number of operations means higher efficiency. Two different programs with different logic can deliver the same output but the efficient one will accomplish the task in lesser number of operations.

Comparison and swapping are the major operations in bubble sort and played major role in efficiency calculation (Complexity). To calculate the number of operations in bubble sort let us keep the above example

Operations in 1st Pass (for an Array of 5 elements)

6	3	5	8	2
3	6	5	8	2
3	6	5	8	2
3	5	6	8	2
3	5	6	8	2
3	5	6	8	2
3	5	6	8	2
3	5	6	2	8

1. Compare 6 and 3
2. 6 is higher, so swap 6 and 3
3. Compare 6 and 5

4. 6 is higher, so swap 6 and 5
5. Compare 6 and 8
6. 8 is higher, so no swap is performed
7. Compare 8 and 2
8. 8 is higher, so swap 8 and 2
9. The largest value, 8, is at the end of the list

Total 04 Comparison + 03 swapping Operations

Operations in 2nd Pass (for an Array of 5 elements)

3	5	6	2	8
3	5	6	2	8
3	5	6	2	8
3	5	6	2	8
3	5	6	2	8
3	5	2	6	8
3	5	2	6	8

1. Compare 3 and 5
2. 5 is higher, so no swap is performed
3. Compare 6 and 5
4. 6 is higher, so no swap is performed
5. Compare 6 and 2
6. 6 is higher, so swap 6 and 2
7. The largest value, 6, bubbled to the appropriate place

Total 03 Comparison + 01 swapping Operations

Operations in 3rd Pass (for an Array of 5 elements)

3	5	2	6	8
3	5	2	6	8
3	5	2	6	8
3	2	5	6	8
3	2	5	6	8

1. Compare 3 and 5
2. 5 is higher, so no swap is performed
3. Compare 5 and 2
4. 5 is higher, so swap 5 and 2
5. The largest value, 5, bubbled to the appropriate place

Total 02 Comparison + 01 swapping Operations

Operations in 4th Pass (for an Array of 5 elements)

3	2	5	6	8
2	3	5	6	8
2	3	5	6	8

1. Compare 3 and 2
2. 3 is higher, so swap 3 and 2
3. The largest value, 3, bubbled to the appropriate place

Total 01 Comparison + 01 swapping Operations

And we found out the Array sorted in ascending order i.e. [2,3,5,6,8]

Total number of Comparison for an Array of 5 elements are $(4+3+2+1)$ 10 and in more generalized way, for the Array of N elements the no of comparison operations are:

$$(N-1)+(N-2)+(N-3)+(N-4)+\dots+(2)+(1)= N*(N-1)/2 \text{ i.e near equivalent to } N^2$$

Similarly total Swapping operations are $(3+1+1+1)$ 6 (approximate to N^2) for an array of size 5 elements. For N elements Array, numbers of Swapping operations are in:

Best Case: - Array is already sorted is 0(zero)

Worst Case:- Array is sorted in apposite order is

$$(N-1)+(N-2)+(N-3)+(N-4)+\dots+(2)+(1)= N*(N-1)/2 \text{ i.e near equivalent to } N^2$$

Therefore in Bubble sort, in best case, total number of operations are $N^2+0=N^2$ and

In worst case, total number of operations are $N^2+ N^2 = 2N^2$

Assignment:

1. Show the comparisons and exchanges that would take place in using a bubble sort to put the following data in ascending order. [3, 8, 3, 2, 7, 5]
2. What changes would have to be made to the bubble_sort function in order to make it sort values in descending order?
3. A variation of the bubble sort is the *cocktail shaker sort* in which, on odd-numbered passes, large values are carried to the top of the list. On even-numbered passes, small values are carried to the bottom of the list. Show the first two passes on the following data. [2, 9, 4, 6, 1, 7]

Insertion Sort

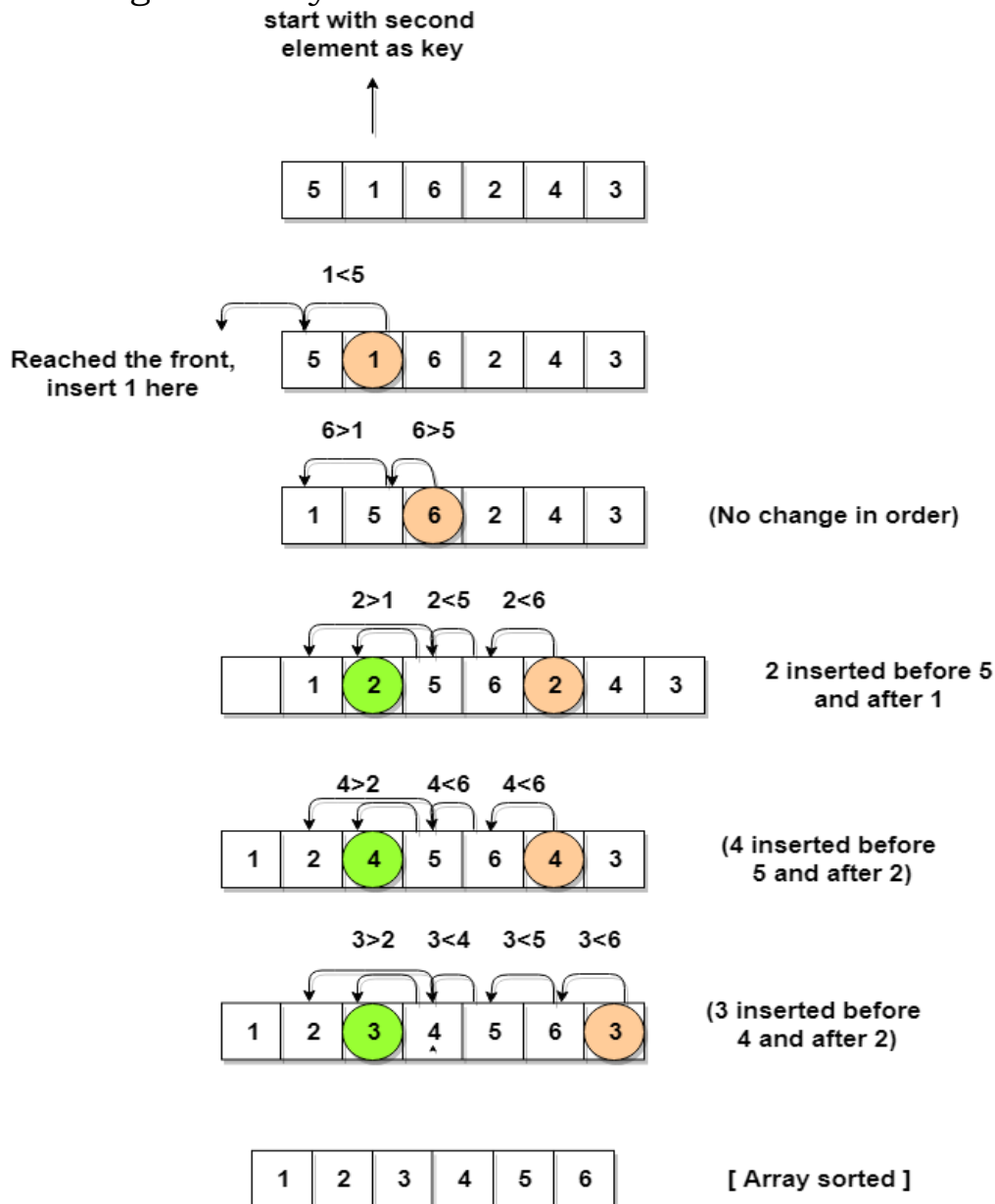
Sorting in computer term means arranging elements in a specific order i.e. either increasing order or decreasing order. Insertion sort is one of the technique to arrange elements in a list.

Following are the steps involved in insertion sort:

- i. We start by making the second element of the given list, i.e. element at index 1, the key.
- ii. We compare the key element with the element(s) before it, in this case, element at index 0:
 - a. If the key element is less than the first element, we insert the key element before the first element.
 - b. If the key element is greater than the first element, then we insert it after the first element.
- iii. Then, we make the third element of the array as key and will compare it with elements to its left and insert it at the right position.
- iv. And we go on repeating this, until the array is sorted.

Let's consider an array with values {5, 1, 6, 2, 4, 3}

Below, we have a pictorial representation of how bubble sort will sort the given array.



Implementation in Python:

```

Arr1 = [20, 12, 18, 25, 6, 35]
Print ("Original List is:", Arr1)
For i in range(1, len(Arr1)):
    Key = Arr1[i]

```

```

    j = i - 1
    While j >=0 and key < Arr1[j]:
        Arr1[j+1] = Arr1[j]
        j = j - 1
    Else:
        Arr1[j+1] = key
    Print ("List after sorting:", Arr1)

```

Output:

```

Original list is : [20, 12, 18, 25, 6, 35]
List After Sorting: [6, 12, 18, 20, 25, 35]

```

Following are some of the important **characteristics of Insertion Sort**:

1. It is efficient for smaller data sets, but very inefficient for larger lists.
2. Insertion Sort is adaptive, that means it reduces its total number of steps if a partially sorted array is provided as input, making it efficient.
3. It is better than Bubble Sort algorithms.
4. Its space complexity is less. Like bubble Sort, insertion sort also requires a single additional memory space.
5. It is a **stable** sorting technique, as it does not change the relative order of elements which are equal.

Number of Operations

There are two operations carried out in the insertion operation i.e. Comparison and exchange operation. Let calculate number of operations of each type for a sequence having N number of elements.

i) The number of comparison:

If we carefully go through the code, we will find that

- During the first iteration of outer loop, in the inner loop there is 01 comparison.
- During the second iteration of outer loop, there are at most 02 comparisons.
- During the N-1 th iteration of outer loop, there are at most N-1 comparisons.

Thus maximum number of comparisons:

$$1+2+ \dots + (N-2) + (N-1) = (N \times (N-1))/2 = (N^2-N)/2 < N^2$$

That means there can be maximum of N^2 comparisons in insertion sort.

ii) The Number of exchange:

If we carefully go through the code, we will find that

- During the first iteration of outer loop, in the inner loop there is 01 exchange.
- During the second iteration of outer loop, there are at most 02 exchanges.
- During the N-1 th iteration of outer loop, there are at most N-1 exchanges.

Thus maximum number of exchanges:

$$1+2+ \dots + (N-2) + (N-1) = (N \times (N-1))/2 = (N^2-N)/2 < N^2$$

That means there can be maximum of N^2 comparisons in insertion sort.

The number of comparisons and exchanges depends on the sequence to be sorted, that is insertion sort is sensitive to the input sequence.

There are 03 possibilities:

- In the best case, when the sequence is already in desired sorted order, there will maximum N-1 number of comparisons and exchanges.
- In the worst case, there will maximum N^2 number of comparisons and exchanges.

Strings

- Can be declared in single quotes (' ') or double quotes (" ") or triple quotes (""" "" or "" "")
- single Quote = 'one line'
- double Quote = "one line with 'eg. jame's'"
- triple Quote = """ Span multiple lines"""

- Each character can be accessed by using their index. Index start from zero(0).
- The indexes of a String begin from 0 to (length-1) in forward direction and -1,-2,-3.....,-length in backward direction.
- String are “immutable”.
- Immutabe means that we cannot change the value. If we have an instance of the string class, any method you call which seems to modify the value, will actually create a new string.

String Methods

Let our string variable be called ‘var’

- length of string – `len(var)`
- convert to string – `str(var)`
- convert to lowercase – `var.lower()`
- convert to uppercase – `var.upper()`
- Is digit/Is alpha – `var.isdigit()/var.isalpha()`
- Replace characters – `var.replace(old,new)`
- Split a sentence – `var.split(delimiter)`
- Swap case – `var.swapcase()`
- Range slice – `var [start index : end index]`

String Concatenation and Formatting

- Concatenation
 - Combining of strings is done by using the (+) operator between them.
 - In order to combine a string with a non-string variable, use str() method to convert non-string to string.

Eg. “tea”+”pot”
Will result into “teapot”

- Formatting
 - The String format operator is %
 - %c - character
 - %s – string conversion via str() prior to formatting
 - %d – signed decimal integer
 - %x %X – hexadecimal integer (lowercase/uppercase)
 - %f – floating point real number

Concatenated string with uncommon characters in python

Prob: Two strings are given you have to modify 1st string such that all the common characters of the 2nd string have to be removed and uncommon characters of the 2nd string have to be concatenated with uncommon characters of the 1st string.

Eg. Input : S1 = “aacdb”
 S2 = “gafd”

Output : “cbgf”

Solution:-

```
Srt1='Geeta' , str2 = 'Babita'
Set1= set(str1)                # convert both string into set
Set2= set(str2)

Common = list(set1 & set2)      # intersection of two set
result = [ch for ch in str1 if ch not common]
        + \
        [ch for ch in str2 if ch not in common]
Print “. Join(result)
```

String Comparison

Python compares two strings through relational operators (<, <=, >, >=, ==, !=) using character-by-character comparison of their Unicode values.

Python Program to count number of vowels using sets in given string

```
str = “implementation”
count = 0
vowel = set (“aeiouAEIOU”)
for alphabet in str:
    if alphabet in vowel:
        count= count + 1
Print (“no. of vowels :”, count)
```

String slices

Part of a string containing some contiguous characters from the string

Say we have a string namely word storing a string “amazing” i.e

	0	1	2	3	4	5	6
a	m	a	z	i	n	g	

Word

-7 -6 -5 -4 -3 -2 -1

Then word [0:3] will give ‘ama’

word [-5:-1] will give ‘azin’

word [2:5] will give ‘azi’

In a string slice you give the slicing range in the form [<begin-index> : <last>]

If, however you skip either of the begin-index or last, python will consider limit of string i.e for missing begin-index it will consider 0 and for missing last value it will consider length of the string.

String Comparison

Program that reads a line & a substring. It should then display the number of occurrences of the given substring in the line.

Sol :

```
line = input ("Enter line")
sub = input ("Enter substring")
length = len (line)
lensub = len(sub)
start = count = 0
end = length
while True:
    Pos = line.find (sub, start, end)
    if Pos!= -1:
        count += 1
```

```

    start = Pos + lensub
else:
    break
if start >= length:
    break
print("No. of occurrences of", sub,':', count)

```

Programs

1. Programs that reads a line & Prints its statistics like:

Number of uppercase letters:

Number of lowercase letters:

Number of alphabets:

Number of digits:

```
Line = input("Enter a line")
```

```
lowercount = uppercount = 0
```

```
digitcount = alphacount = 0
```

```
for a in line:
```

```
    if a.islower() :
```

```
        lowercount += 1
```

```
    elif a.isupper() :
```

```
        uppercount += 1
```

```
    elif a.isdigit() :
```

```
        digitcount += 1
```

```
    elif a.isalpha() :
```

```
        alphacount += 1
```

```
print (" No. of uppercase letters:" uppercount)
```

```
print (" No. of lowercase letters:" lowercount)
```

```
print (" No. of alphabets:" alphacount)
```

```
print (" No. of digits:" digitcount)
```

2. Program that reads a string & checks whether it is a palindrome string or not.

Solution:-

```
string = input ("Enter a string:")
```

```

length = len (string)
Mid = length/2
rev = -1
for a in range (Mid) :
    if string [a] == string [rev]
        a += 1
        rev -= 1
    else:
        Print (string," is not a palindrome")
        break
    else:
        Print (string," is a palindrome")

```

3. Write a program that takes a string with multiple words & then capitalizes the first letter of each word & forms a new string out of it.

Solution:-

```

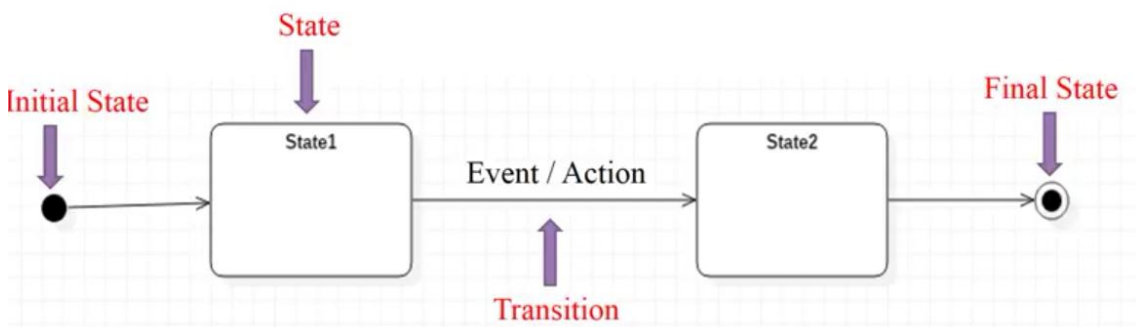
String = input(" Enter a string:")
length = len (string)
a = 0
end = length
string2 = ''
while a < length:
    if a == 0:
        string2 += string [0].upper( )
        a += 1
    elif (string [a] == ' ' and string [a + 1]!=""):
        string2 += string [a]
        string2 += string [a + 1]. upper( )
        a += 2
    else:
        string2 += string [a]
        a += 1
Print (" original string : " , string)
Print (" capitalized words string " , string2)

```

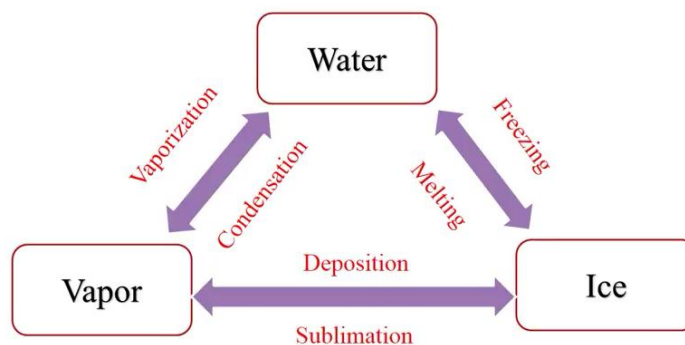
State-Transition Diagrams

State-transition diagrams describe all of the states that an object can have, the events under which an object changes state (transitions), the conditions that must be fulfilled before the transition will occur (guards), and the activities undertaken during the life of an object (actions). State-transition diagrams are very useful for describing the behavior of individual objects over the full set of use cases that affect those objects. State-transition diagrams are not useful for describing the collaboration between objects that cause the transitions.

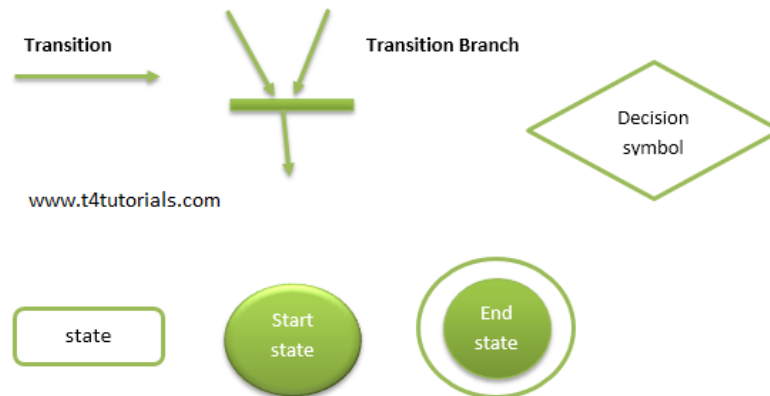
NOTATIONS



SIMPLE STATE DIAGRAM



State transition diagram symbols



Example of State Transition Diagram:

You need to develop a web-based application in such a way that user can search other users, and after getting search complete, the user can send the friend request to other users. If the request is accepted, then both users are added to the friend list of each other. If one user does not accept the friend request. The second user can send another friend request. The user can also block each other.

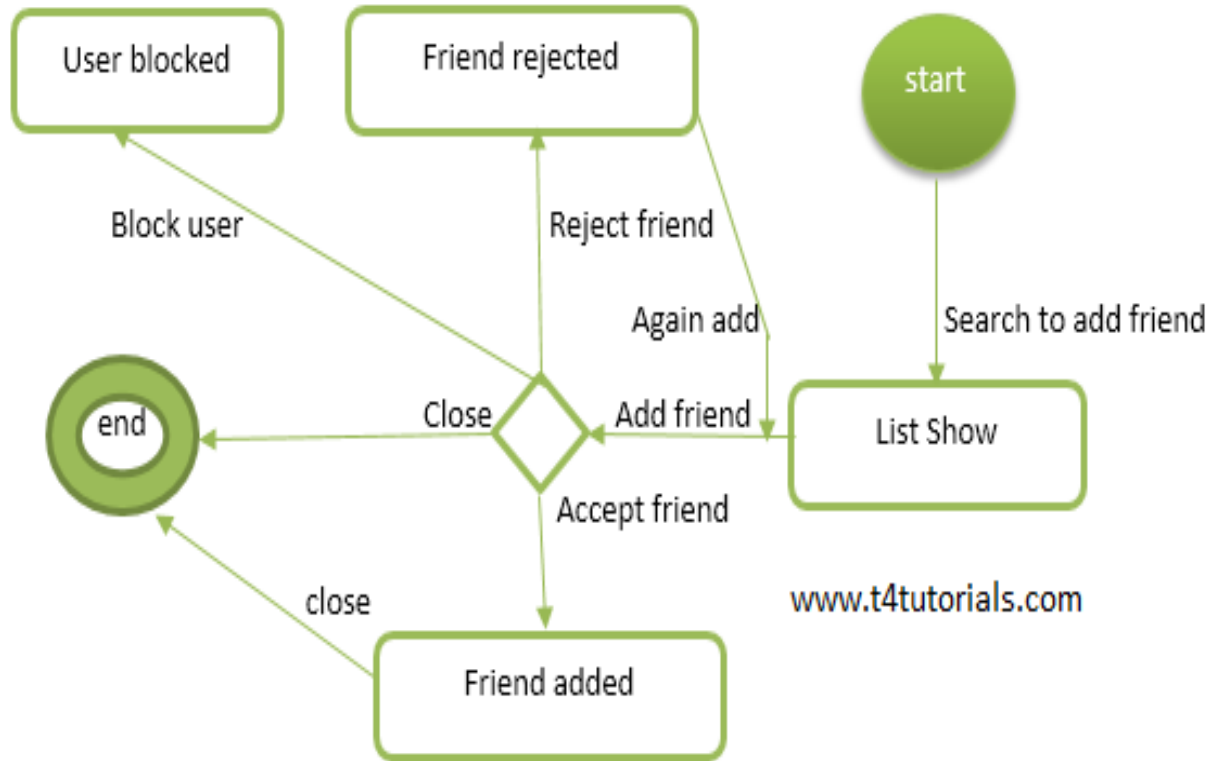
Solution:

1. First of all, identify the object that you will create during the development of classes in oop
2. Identity the actions or events
3. Identify the possible states for object
4. Draw the diagram.

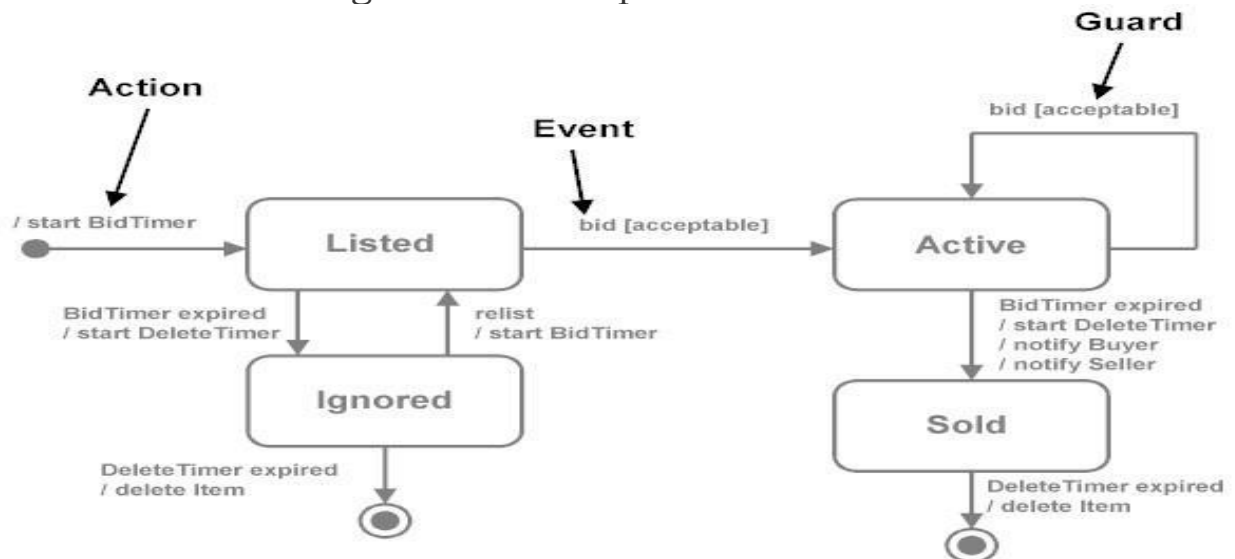
Object: friends

Events or actions: Search to add a friend, add a friend, accept a friend, reject a friend, again add, block user and close.

States: Start, the friend added, friend rejected, user blocked and end.



State transition diagrams for BID process is shown below:



- State.** A condition during the life of an object in which it satisfies some condition, performs some action, or waits for some event.

- **Event.** An occurrence that may trigger a state transition. Event types include an explicit signal from outside the system, an invocation from inside the system, the passage of a designated period of time, or a designated condition becoming true.
- **Guard.** A boolean expression which, if true, enables an event to cause a transition.
- **Transition.** The change of state within an object.
- **Action.** One or more actions taken by an object in response to a state change.

State Transition diagram for a process

As a process executes, it changes state. The state of a process is defined in part by the current activity of that process.

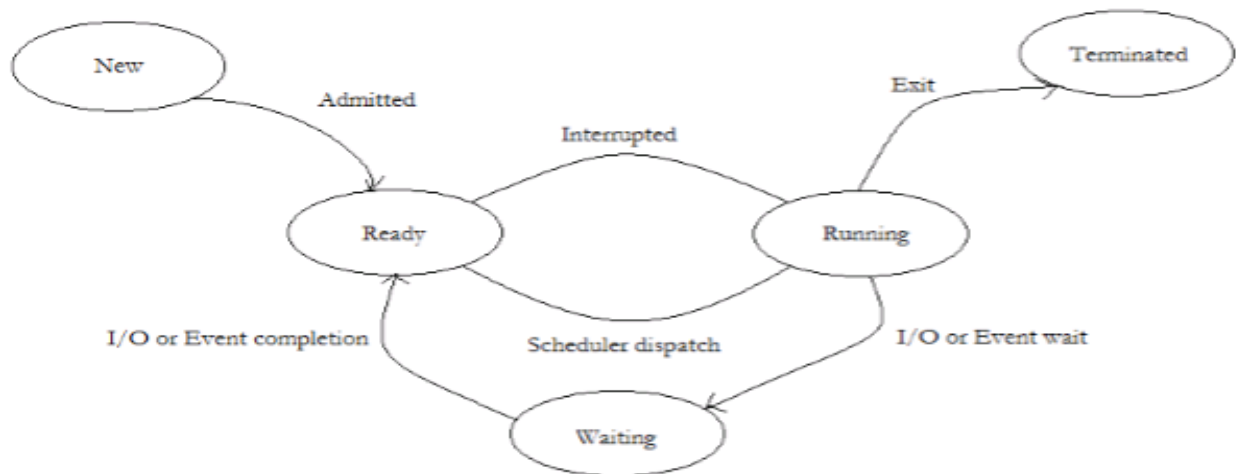


Fig: Process state transition diagram.

- Process can have one of the following five states at a time.
 - 1. New state:** A process that just has been created but has not yet been admitted to the pool of execution processes by the operating system. Every new operation which is requested to the system is known as the new born process.
 - 2. Ready state:** When the process is ready to execute but he is waiting for the CPU to execute then this is called as the ready state. After completion of the input and output the

process will be on ready state means the process will wait for the processor to execute.

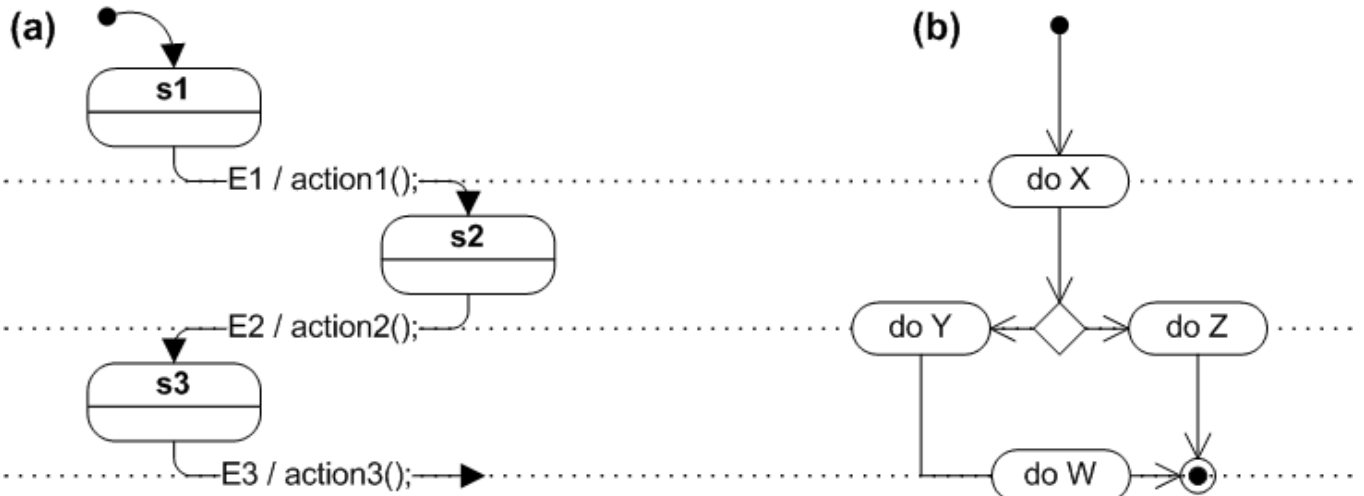
3. Running state: The process that is currently being executed. When the process is running under the CPU, or when the program is executed by the CPU, then this is called as the running process and when a process is running then this will also provide us some outputs on the screen.

4. Waiting or blocked state: A process that cannot execute until some event occurs or an I/O completion. When a process is waiting for some input and output operations then this is called as the waiting state and in this process is not under the execution instead the process is stored out of memory and when the user will provide the input and then this will again be on ready state.

5. Terminated state: After the completion of the process, the process will be automatically terminated by the CPU. So this is also called as the terminated state of the process. After executing the complete process the processor will also deallocate the memory which is allocated to the process. So this is called as the terminated process.

State diagrams versus flowcharts

Newcomers to the state machine formalism often confuse **state diagrams** with **flowcharts**. The figure below shows a comparison of a **state diagram** with a flowchart. A state machine (panel (a)) performs actions in response to explicit events. In contrast, the flowchart (panel (b)) does not need explicit events but rather



transitions from node to node in its graph automatically upon completion of activities.^[9]

Nodes of flowcharts are edges in the induced graph of states. The reason is that each node in a flowchart represents a program

command. A program command is an action to be executed. So it is not a state, but when applied to the program's state, it results in a transition to another state.

In more detail, the source code listing represents a program graph. Executing the program graph (parsing and interpreting) results in a state graph. So each program graph induces a state graph. Conversion of the program graph to its associated state graph is called "unfolding" of the program graph.

The program graph is a sequence of commands. If no variables exist, then the state consists only of the program counter, which keeps track of where in the program we are during execution (what is the next command to be applied).

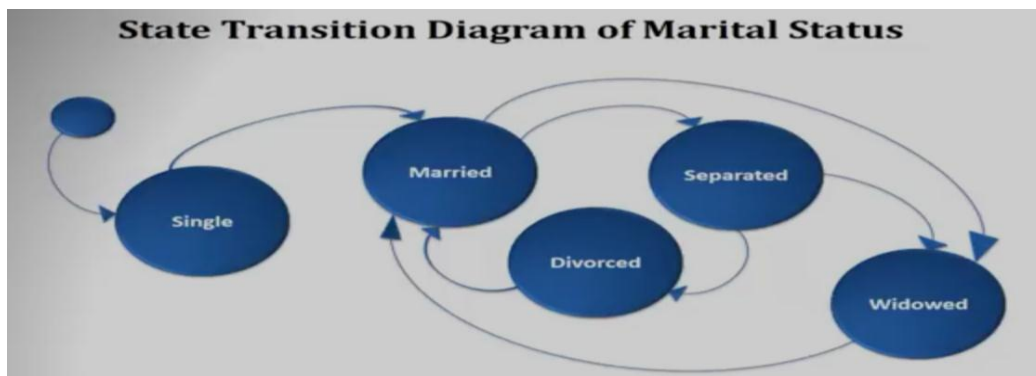
In this case before executing a command the program counter is at some position (state before the command is executed).

Executing the command moves the program counter to the next command. Since the program counter is the whole state, it follows that executing the command changed the state. So the command itself corresponds to a transition between the two states.

Now consider the full case, when variables exist and are affected by the program commands being executed. Then between different program counter locations, not only does the program counter change, but variables might also change values, due to the commands executed. Consequently, even if we revisit some program command (e.g. in a loop), this doesn't imply the program is in the same state.

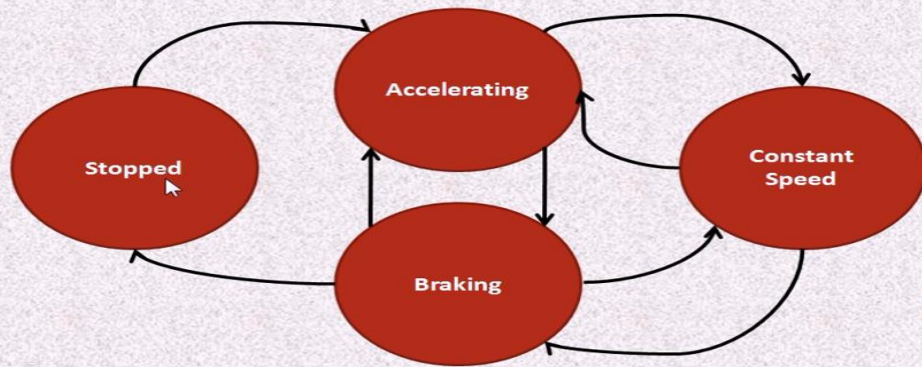
More Examples on STD

Example-1

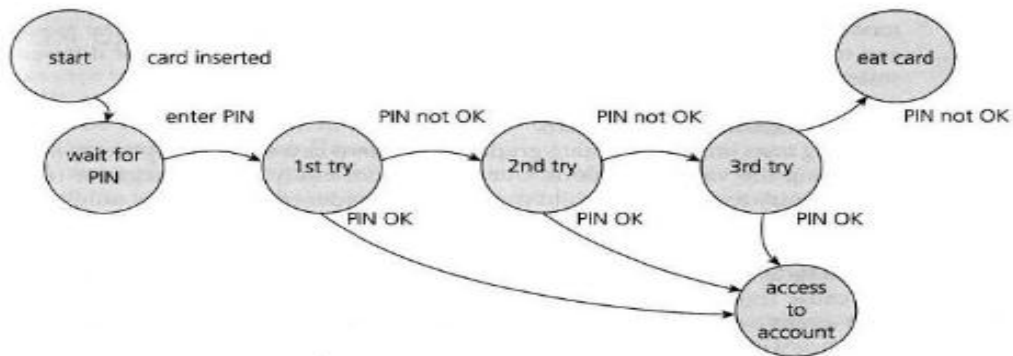


STATE – EXAMPLE 2 – CAR

- States are Stopped, Accelerating, Constant Speed and Decelerating

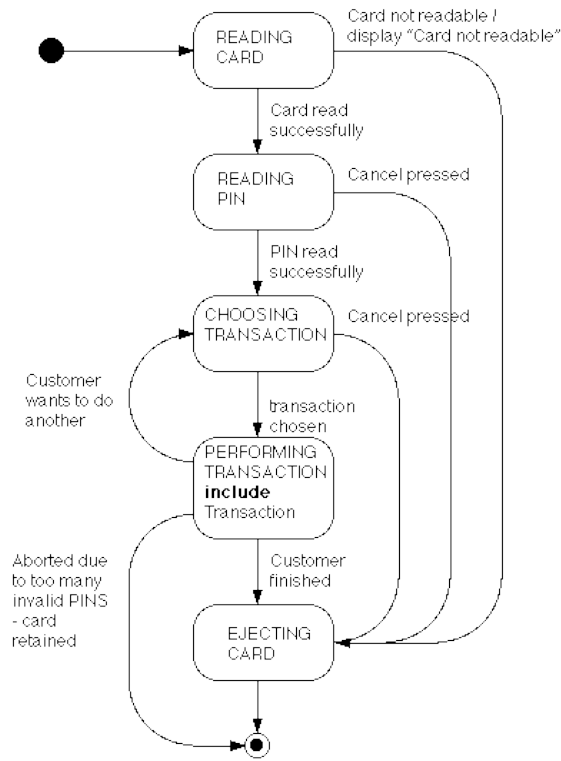


Checking PIN/Password (at most 3 Chances)

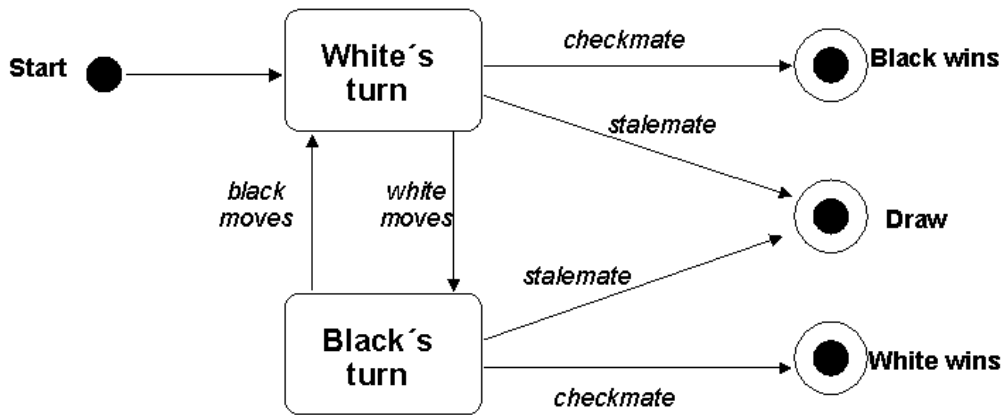


state transition diagrams.for ATM

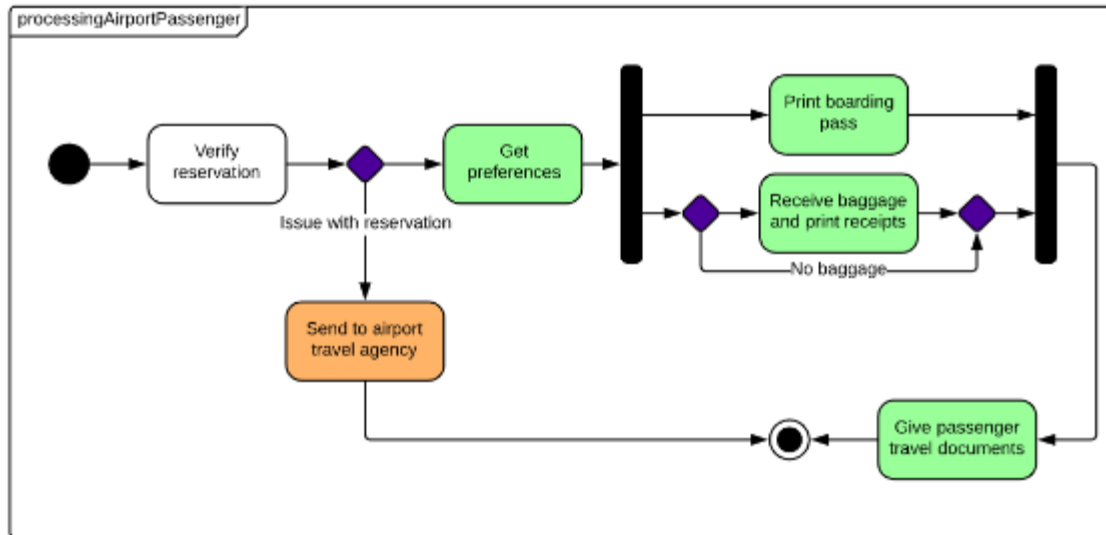
State-Chart for One Session



Chess game



Airport check-in state diagram example



Advantages :

- Allows testers to familiarise with the software design and enables them to design tests effectively.
- This testing technique will provide a pictorial or tabular representation of system behavior which will make the tester to cover and understand the system behavior effectively.
- It also enables testers to cover the unplanned or invalid states.
- By using this testing, technique tester can verify that all the conditions are covered, and the results are captured

Disadvantages:

- Their biggest limitation is that they are not good at describing behavior that involved several objects. For these cases use an interaction diagram or an activity diagram
- we can't rely in this technique every time. For example, if the system is not a finite system (not in sequential order), this technique cannot be used.

BASIC OF NoSQL DATABASES

NoSQL (Not Only SQL)databases are non-relational databases that do not have strict, rigid schemas and do not have traditional table-

format based data model to store data. They run in clusters and cater to large data sets scalable upto web scale.

A NoSQL database has dynamic schema for unstructured data, and data is stored in many ways: it can be column-oriented, document-oriented, graph-based or organized as a Key/Value store. This flexibility means that:

- You can create documents without having to first define their structure
- Each document can have its own unique structure
- The syntax can vary from database to database, and
- You can add fields as you go.

In most situations, SQL databases are vertically scalable, which means that you can increase the load on a single server by increasing things like CPU, RAM or SSD. NoSQL databases, on the other hand, are horizontally scalable. This means that you handle more traffic by sharding, or adding more servers in your NoSQL database. It's like adding more floors to the same building versus adding more buildings to the neighborhood. The latter can ultimately become larger and more powerful, making NoSQL databases the preferred choice for large or ever-changing data sets. Some examples of SQL databases include MySQL, Oracle, PostgreSQL, and Microsoft SQL Server. NoSQL database examples include MongoDB, BigTable, Redis, RavenDB, Cassandra, HBase, Neo4j and CouchDB.

RDBMS vs NoSQL

RDBMS

- Structured and organized data
- Structured query language (SQL)
- Data and its relationships are stored in separate tables.

- Data Manipulation Language, Data Definition Language
- Tight Consistency

NoSQL

- Stands for Not Only SQL
- No declarative query language
- No predefined schema
- Key-Value pair storage, Column Store, Document Store, Graph databases
- Eventual consistency rather ACID property
- Unstructured and unpredictable data
- CAP (Consistency, Availability and Partition tolerance) Theorem
- Prioritizes high performance, high availability and scalability
- BASE (**B**asically **A**vailable **S**oft state **E**ventual consistency)
Transaction

Hence NoSQL databases are developed to :

- Provide support to thousands to millions of concurrent users of modern applications (such as Facebook, Amazon Online Store, Google Earth etc)
- Deliver very fast response time to globally distributed users.
- Handle all types of data without any boundation of structure
- Provide rapid adaptability to fast changing requirements with frequent updates and new features.
- Provide an always on performance, i.e., no down time.

Advantages of NoSQL databases include:

- Flexible Data Model
- High scalability
- Distributed Computing
- Lower cost
- Schema flexibility, semi-structure data

- No complicated Relationships
- High Performance
- Open Source

Disadvantages

- Lack of standardization
- Limited query capabilities (so far)
- Consistency
- Backup of database

Types of NoSQL databases :

- (i) Key-value databases
- (ii) Document databases
- (iii) Column family stores databases
- (iv) Graph databases

Key-value databases:

- Designed to handle huge amounts of data.
- Based on Amazon's Dynamo paper.
- Key value stores allow developer to store schema-less data.
- In the key-value storage, database stores data as hash table where each key is unique and the value can be string, JSON, BLOB (Binary Large Objec) etc.
- A key may be strings, hashes, lists, sets, sorted sets and values are stored against these keys.
- For example a key-value pair might consist of a key like "Name" that is associated with a value like "Robin".
- Key-Value stores can be used as collections, dictionaries, associative arrays etc.
- Key-Value stores follow the 'Availability' and 'Partition' aspects of CAP theorem.
- Key-Values stores would work well for shopping cart contents, or individual values like color schemes, a landing page URI, or a default account number.

Example of Key-value store DataBase : Cassandra, Redis, Dynamo, Riak etc.

Pictorial Presentation :



Document databases :

- A collection of documents
- Data in this model is stored inside documents.
- A document is a key value collection where the key allows access to its value.
- Documents are not typically forced to have a schema and therefore are flexible and easy to change.
- Documents are stored into collections in order to group different kinds of data.
- Documents can contain many different key-value pairs, or key-array pairs, or even nested documents.

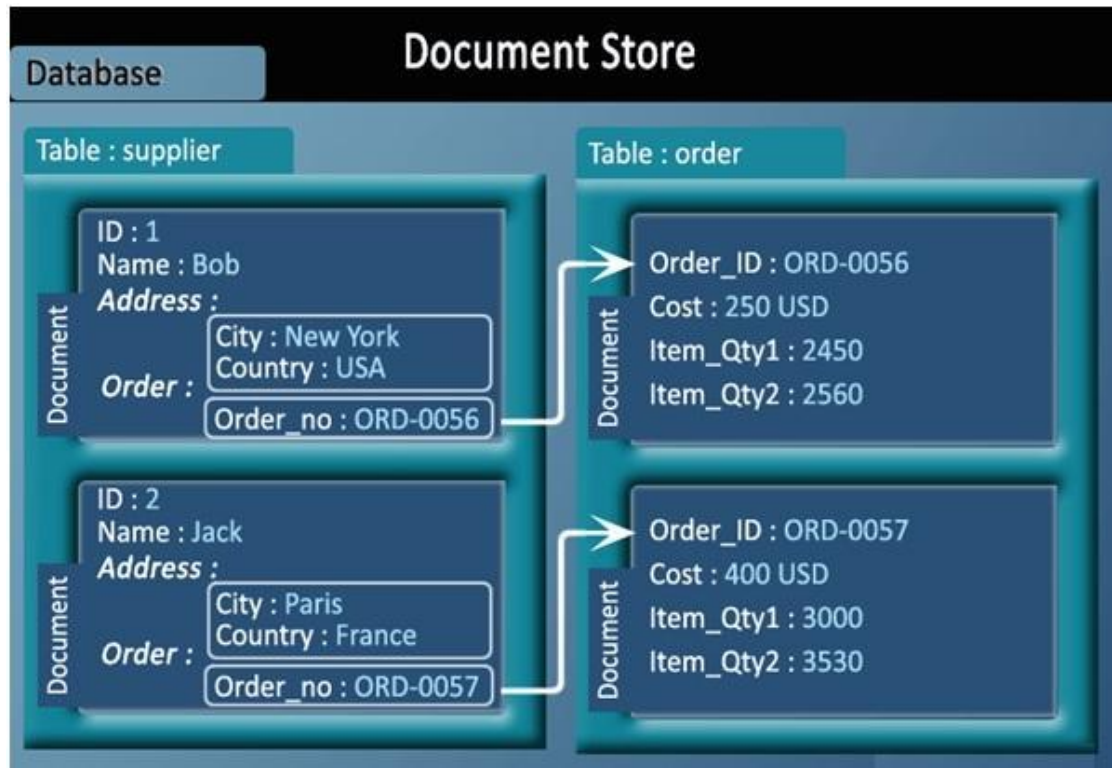
Here is a comparison between the classic relational model and the document model :

Relational model	Document model
Tables	Collections
Rows	Documents

Columns	Key/value pairs
Joins	not available

Example of Document Oriented databases : MongoDB, CouchDB, DocumentDB etc.

Pictorial Presentation :



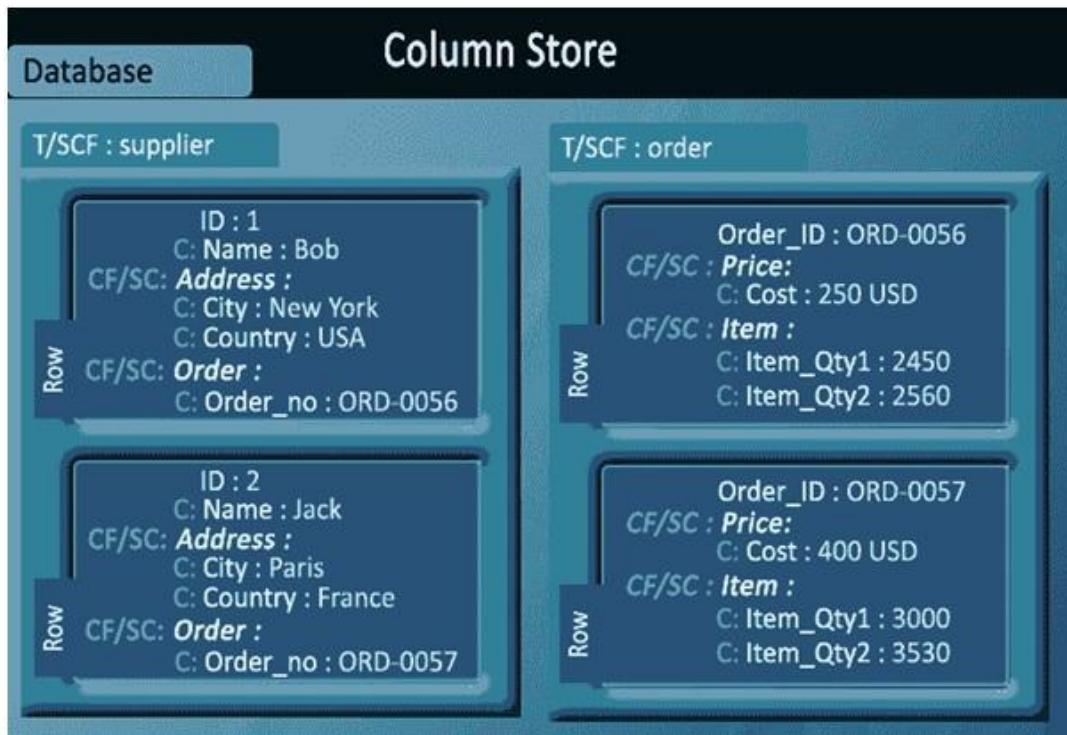
Column family stores databases:

- Column-oriented databases primarily work on columns and every column is treated individually.
- Values of a single column are stored contiguously.
- Column stores data in column specific files.
- In Column stores, query processors work on columns too.
- All data within each column datafile have the same type which makes it ideal for compression.
- Column stores can improve the performance of queries as it can access specific column data.
- High performance on aggregation queries (e.g. COUNT, SUM, AVG, MIN, MAX).

- Works on data warehouses and business intelligence, customer relationship management (CRM), Library card catalogs etc.

Example of Column-oriented databases : BigTable, Cassandra, SimpleDB etc.

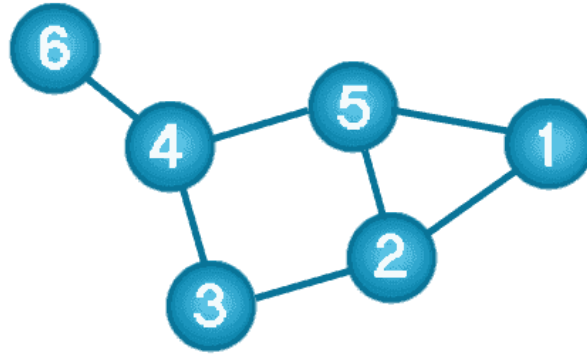
Pictorial Presentation :



Graph databases:

A graph data structure consists of a finite (and possibly mutable) set of ordered pairs, called edges or arcs, of certain entities called nodes or vertices.

The following picture presents a labeled graph of 6 vertices and 7 edges.



What is a Graph Databases?

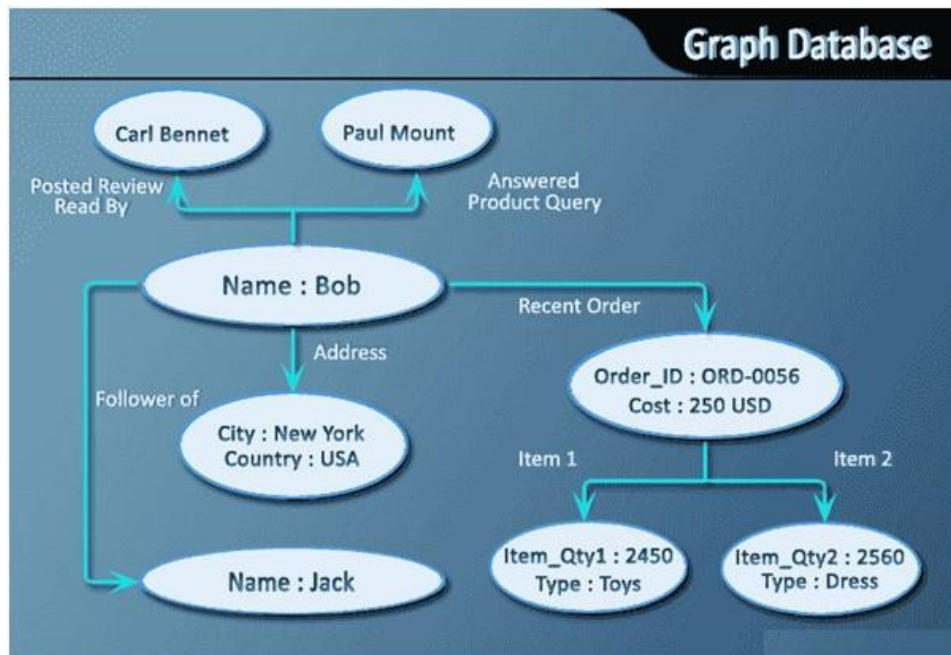
- A graph database stores data in a graph.
- It is capable of elegantly representing any kind of data in a highly accessible way.
- A graph database is a collection of nodes and edges
- Each node represents an entity (such as a student or business) and each edge represents a connection or relationship between two nodes.
- Every node and edge are defined by a unique identifier.
- Each node knows its adjacent nodes.
- As the number of nodes increases, the cost of a local step (or hop) remains the same.
- Index for lookups.

Here is a comparison between the classic relational model and the graph model :

Relational model	Graph model
Tables	Vertices and Edges set
Rows	Vertices
Columns	Key/value pairs
Joins	Edges

Example of Graph databases : OrientDB, Neo4J, Titan.etc.

Pictorial Presentation :



SQL vs NoSQL: High-Level Differences

- SQL databases are primarily called as Relational Databases (RDBMS); whereas NoSQL database are primarily called as non-relational or distributed database.
- SQL databases are table based databases whereas NoSQL databases are document based, key-value pairs, graph databases or wide-column stores. This means that SQL databases represent data in form of tables which consists of n number of rows of data whereas NoSQL databases are the collection of key-value pair, documents, graph databases or wide-column stores which do not have standard schema definitions which it needs to adhered to.
- SQL databases have predefined schema whereas NoSQL databases have dynamic schema for unstructured data.
- SQL databases are vertically scalable whereas the NoSQL databases are horizontally scalable. SQL databases are scaled by increasing the horse-power of the hardware. NoSQL databases are scaled by increasing the databases servers in the pool of resources to reduce the load.

- SQL databases use SQL (structured query language) for defining and manipulating the data, which is very powerful. In NoSQL database, queries are focused on collection of documents. Sometimes it is also called as UnQL (Unstructured Query Language). The syntax of using UnQL varies from database to database.
- SQL database examples: MySql, Oracle, Sqlite, Postgres and MS-SQL. NoSQL database examples: MongoDB, BigTable, Redis, RavenDb, Cassandra, Hbase, Neo4j and CouchDb
- For complex queries: SQL databases are good fit for the complex query intensive environment whereas NoSQL databases are not good fit for complex queries. On a high-level, NoSQL don't have standard interfaces to perform complex queries, and the queries themselves in NoSQL are not as powerful as SQL query language.
- For the type of data to be stored: SQL databases are not best fit for hierarchical data storage. But, NoSQL database fits better for the hierarchical data storage as it follows the key-value pair way of storing data similar to JSON data. NoSQL database are highly preferred for large data set (i.e for big data). Hbase is an example for this purpose.
- For scalability: In most typical situations, SQL databases are vertically scalable. You can manage increasing load by increasing the CPU, RAM, SSD, etc, on a single server. On the other hand, NoSQL databases are horizontally scalable. You can just add few more servers easily in your NoSQL database infrastructure to handle the large traffic.
- For high transactional based application: SQL databases are best fit for heavy duty transactional type applications, as it is more stable and promises the atomicity as well as integrity of the data. While you can use NoSQL for transactions purpose, it is still not comparable and stable enough in high load and for complex transactional applications.
- For support: Excellent support are available for all SQL database from their vendors. There are also lot of independent

consultations who can help you with SQL database for a very large scale deployments. For some NoSQL database you still have to rely on community support, and only limited outside experts are available for you to setup and deploy your large scale NoSQL deployments.

- For properties: SQL databases emphasizes on ACID properties (Atomicity, Consistency, Isolation and Durability) whereas the NoSQL database follows the Brewers CAP theorem (Consistency, Availability and Partition tolerance)
- For DB types: On a high-level, we can classify SQL databases as either open-source or close-sourced from commercial vendors. NoSQL databases can be classified on the basis of way of storing data as graph databases, key-value store databases, document store databases, column store database and XML databases.

Production deployment

There is a large number of companies using NoSQL like :

- Google
- Facebook
- Mozilla
- Adobe
- Foursquare
- LinkedIn
- Digg
- McGraw-Hill Education
- Vermont Public Radio

Introduction

What is MongoDB?

MongoDB is an open-source document database and leading NoSQL database. MongoDB is written in C++.

MongoDB is a cross-platform, document oriented database

It provides:

- high performance
- high availability
- easy scalability.

MongoDB works on concept of collection and document.

What is Document oriented database?

It is a modern way of storing data in a different way in place of using row / column method. It gives a flexibility to the user/programmer/data base administrator to work on a variety of data, without actually deciding a predefined schema.

What is NoSQL database?

It stands for “**Not Only SQL**” database. NoSQL is an approach to database design that can accommodate a wide variety of data models, including key-value, document, columnar and graph formats. NoSQL, which stand for "not only SQL," is an alternative to traditional relational databases in which data is placed in tables and data schema is carefully designed before the database is built. NoSQL databases are especially useful for working with large sets of distributed data.

General Difference

RDBMS	Document database or NoSQL or MongoDB									
It contains rows and columns	It contains key-value, document concept of storing the data									
How data is stored in RDBMS Eg.: <table border="1" data-bbox="191 1522 781 1654"><thead><tr><th colspan="3">Data</th></tr><tr><th>S No</th><th>Name</th><th>Age</th></tr></thead><tbody><tr><td>1</td><td>Rajesh</td><td>21</td></tr></tbody></table>	Data			S No	Name	Age	1	Rajesh	21	{ "sno" : "1", "name" : "Rajesh", "Age" : "21", }
Data										
S No	Name	Age								
1	Rajesh	21								
RDBMS	Document database or NoSQL or MongoDB									
If only one additional value is	Being flexible, has a vast scope									

Dynamic Schema	Dynamic schema means that documents in the same collection do not need to have the same set of fields or structure, and common fields in a collection's documents may hold different types of data.
-----------------------	---

Advantages of MongoDB over RDBMS

- Schema less
- Structure of a single object is clear.
- No complex joins.
- Deep query-ability. MongoDB supports dynamic queries on documents using a document-based query language that's nearly as powerful as SQL.
- Tuning. (improving the performance of database better)
- Ease of scale-out – MongoDB is easy to scale.
- Conversion/mapping of application objects to database objects not needed.
- Uses internal memory for storing the (windowed) working set, enabling faster access of data.

Why Use MongoDB?

- Document Oriented Storage
- Index on any attribute
- Replication and high availability
- Auto-sharding (when data grows one machine may not be enough to handle so it has the capability of working with data stored at multiple machine)
- Rich queries
- Fast in-place updates
- Professional support by MongoDB

Where to Use MongoDB?

- Big Data

- Content Management and Delivery
- Mobile and Social Infrastructure
- User Data Management
- Data Hub

Installing MongoDB

→ It is very easy to install MongoDB, just download the installer file from the internet or from and install it. But do remember before running the server of MongoDB to make use of the same do the following steps:

- Create a folder in root directory with the name “data”
- Inside the “data” folder create following two folders
 - A folder named “db” – for database transactions
 - A folder named “log” – for keeping log of different transactions/activities

Once the above folders are created switch to MongoDB folder which is given at the time to installation and open file “**mongod.exe**” to execute the server.

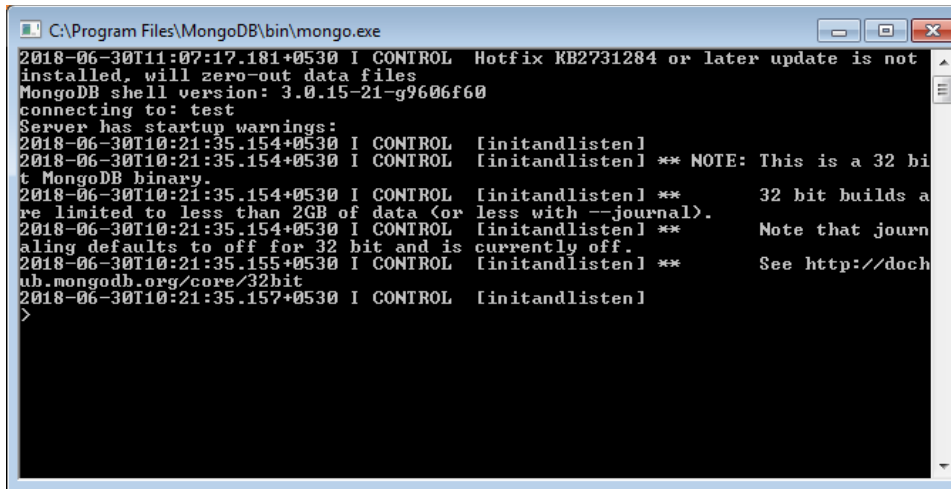
Note: Windows 7 supports MongoDB 3.0 where as any other version higher than windows 7 supports higher version than MongoDB 3.0 (like MongoDB 3.2 etc.)

How to find whether server is ready: when the following window appears containing last line as “connections now open” it means the server is ready to work.

```

C:\Program Files\MongoDB\bin\mongod.exe
2018-06-30T10:21:35.154+0530 I CONTROL [initandlisten]
2018-06-30T10:21:35.154+0530 I CONTROL [initandlisten] ** NOTE: This is a 32 bit
MongoDB binary.
2018-06-30T10:21:35.154+0530 I CONTROL [initandlisten] **      32 bit builds a
re limited to less than 2GB of data (or less with --journal).
2018-06-30T10:21:35.154+0530 I CONTROL [initandlisten] **      Note that journ
aling defaults to off for 32 bit and is currently off.
2018-06-30T10:21:35.155+0530 I CONTROL [initandlisten] **      See http://doch
ub.mongodb.org/core/32bit
2018-06-30T10:21:35.157+0530 I CONTROL [initandlisten]
2018-06-30T10:21:35.162+0530 I INDEX [initandlisten] allocating new ns file C
:\data\db\local.ns, filling with zeroes...
2018-06-30T10:21:35.571+0530 I STORAGE [FileAllocator] allocating new datafile
C:\data\db\local.0, filling with zeroes...
2018-06-30T10:21:35.573+0530 I STORAGE [FileAllocator] creating directory C:\da
ta\db\_tmp
2018-06-30T10:21:35.907+0530 I STORAGE [FileAllocator] done allocating datafile
C:\data\db\local.0, size: 64MB, took 0.33 secs
2018-06-30T10:21:35.914+0530 I NETWORK [initandlisten] waiting for connections
on port 27017
2018-06-30T10:22:32.440+0530 I NETWORK [initandlisten] connection accepted from
127.0.0.1:49549 #1 (1 connection now open)
2018-06-30T10:22:59.205+0530 I NETWORK [conn1] end connection 127.0.0.1:49549 <
0 connections now open>
  
```

Now minimize this window and open “**mongo.exe**” from the MongoDB installed folder that will show the following window



Here “>” is known as MongoDB prompt, from where commands can be supplied to create database, collections, documents etc.

Data types in MongoDB : Few of the supported data types are: (**S**
No 1 to 7 is for students)

S No	Data type	Description
1	String	most commonly used datatype to store the data. String in MongoDB must be UTF-8 valid (UTF-8 is a compromise character encoding that can be as compact as ASCII (if the file is just plain English text) but can also contain any unicode characters (with some increase in file size). UTF stands for Unicode Transformation Format . The '8' means it uses 8 -bit blocks to represent a character.)
2	Integer	to store a numerical value. Integer can be 32 bit or 64 bit depending upon your server.
3	Double	to store floating point values
4	Boolean	to store a boolean (true/ false) value.
5	Timestamp	timestamp. This can be handy for recording when a document has been modified or added.
6	Date	to store the current date or time in UNIX time format (January 1st 1970 00:00:00). You can specify your own date time by creating object of

		Date and passing day, month, year into it.
7	ObjectId	to store the document's ID.
8	Arrays	to store arrays or list or multiple values into one key.
9	Object	for embedded documents (primary key/relationships)
10	Null	to store a Null value.
11	Binary data	to store binary data (audio/video files etc)
12	Code	to store JavaScript code into the document
13	Regular expression	to store regular expression
14	Min/ Max keys	to compare a value against the lowest and highest BSON elements
15	Symbol	identical to a string; however, it's generally reserved for languages that use a specific symbol type.

Working with MongoDB

Database management system gives an option for user / programmer to store its data to the computer which actually stores the data in either of the following 2 forms

1. Using data base files: which contains fields and a set of values stored in different fields collectively known as a record
2. Using relational data base: in which a database contains different tables (entity) where each table is divided into columns (attributes) and rows (tuple) . Different tables can be connected with each other as per the requirement by using relationship concept of keys (primary/foreign).

In the above two ways there is a fix schema of database and the user/programmer has to abide to the same. One cannot add extra value at run time without actually changing the schema, to overcome this problem MongoDB gives a flexible

solution of schema-less data base. **Before starting with the commands do remember that it is a case sensitive database (commands, names, values etc).**

Comm and For	Command / instruction With syntax	Purpose / description	Example <i>Italics words shows output/message from MongoDB command analyser</i>
Creating / using database	use <DATABASE_NAME>	use command is used in two ways to create as well as to bring database to use	use MyDB <i>switched to db MyDB will be displayed</i>
View list of database(s)	show databases OR show dbs	For viewing list of existing databases	show databases OR show dbs this will show list of databases eg. local 0.78125GB MyDB 0.00000GB test 0.23012GB
Creating collection to database	db.createCollection (“<collection_name>”)	To add / create a collection to database	db.createCollection(“myColl”) { "ok" : 1 }
View list of collections(s)	show collections	To view list of collections in current database	show collections
Inserting document	db.<Collection_name>.insert ({	→To add documents to the collection (records to database file or	db.myColl.insert ({ “fname”:”Vaibhav”, “lname”:”Jain”,

<p>to collection</p>	<pre> <key : Value pair1>, <key : Value pair2>, , <key : Value pairn> }) OR db.<Collection _name>.save({ <key : Value pair1>, <key : Value pair2>, , <key : Value pairn> }) </pre>	<p>rows to table)</p> <p>→When a document is inserted to a collection an auto generated id (“_id”) gets automatically inserted to the collection (if not specified)</p> <p>→ This command can be used ‘n’ number of times for inserting documents to collection</p>	<pre> “phone”:1234567890, }) WriteResult({ “nInserted:1}) </pre> <p>In example below id can be inserted by user</p> <pre> db.myColl.insert ({ “_id” : 110 “fname”:"Vaibhav", “lname”:"Jain", “phone”:1234567890, }) WriteResult({ “nInserted:1}) </pre>
<p>Viewing documents or querying document</p>	<pre> db.<Collection _name>.find() db.<Collection _name>.find().pretty() </pre>	<p>To view all documents</p> <p>Formatted document</p>	<pre> db.myColl.find() {“_id” : ObjectId{“5b378dce90c313500d328ac9”},“fname” : ”Vaibhav”,“lname” : “Jain”,“phone”:1234567890 } db.myColl.find().pretty() {“_id” : ObjectId{“5b378dce90c313500d328ac9”}, “fname” : ”Vaibhav”, </pre>

			<pre> "lname" : "Jain", "phone":1234567890 } </pre>
	<pre> db.<Collection_name>.find({ "key" : "value" }) </pre>	To view records as per given condition	<pre> db.myColl.find({ "fname" : "Vaibhav" }) </pre>
Deleting document	<pre> db.<Collection_name>.remove([<Delete Criteria>]) </pre>	<p>→To delete a document</p> <p>→In case of same value being repeated multiple times</p> <p>→to truncate all the records</p>	<pre> db.myColl.remove({"fname":"Vaibhav"}) db.myColl.remove({"fname":"Vaibhav"}, 1) db.myColl.remove() </pre>
Updating document	<pre> db.<Collection_name>.update([<update Criteria>, <updated data>]) </pre>	<p>→ for updating all or particular (record) document</p> <p>update() is used</p>	<pre> db.myColl.update({"fname":"Vaibhav" }, { "fname":"Vaibhav", "phone":9944661133 }) </pre>
Dropping Collection	<pre> db.<Collection_name>.drop() </pre>	<p>→ for deleting collection, it should be remembered that in order to delete a particular collection, first the database</p>	<pre> use myDB <i>switched to db MyDB will be displayed</i> db.myColl.drop() <i>true</i> </pre>

		containing your collection to be bring in use drop () returns <i>true</i> when collection is dropped else it returns <i>false</i>	
Dropping Database	db.dropDatabase()	→to delete a database the database to be brought into use	use myDB <i>switched to db MyDB will be displayed</i> db.dropDatabase() { <i>"dropped" : "myDB", "ok" : 1</i> }
Sorting documents	db.<Collection name>.find().sort({ <field>: 1 OR -1 })	→ for arranging the documents either in increasing / decreasing order 1 - increasing order -1 - decreasing order	db.myColl.find().sort({ "fname":1 })

#program to insert values in a list and find largest and smallest number from it

```
arr=[]
```

```
n=int(input('Enter Number of elements'))
```

```
for i in range(0,n):
```

```
    x=int(input('Enter Number'))
```

```

arr.insert(i,x)
l=g=arr[0]
for i in range(0,n):
    if(g<arr[i]):
        g=arr[i]
    if(l>arr[i]):
        l=arr[i]
print("Greater Number=",g)
print("Lesser No=",l)

```

Output: Enter Number of elements4
Enter Number23
Enter Number1
Enter Number56
Enter Number444
Greater Number= 444
Lesser No= 1

Program to find third largest element from a list

```

a=[]
n=int(input('Enter Number of elements'))
for i in range(1,n+1):
    b=int(input('Enter Number'))
    a.append(b)
a.sort()

```

```
print("Third largest element is:",a[n-3])
```

Output:

```
Enter Number of elements      5
Enter Number      22
Enter Number      66
Enter Number      55
Enter Number      77
Enter Number      11
Third largest element is: 55
```

Program to check that a number is prime or not

```
no=int(input("Enter Number : "))
for i in range(2,no):
    ans=no%i
```

```
if ans==0:
    print ('Non Prime')
    break
elif i==no-1:
    print('Prime Number')
```

Output:

Enter Number : 8

Non Prime

>>> ===== RESTART

=====

>>>

Enter Number : 7

Prime Number

Program to check that a string is palindrome or not

```
st=input("Enter Any String ")
s=""
l=len(st)
a=range(l-1,-1,-1)
for i in a:
    s = s + st[i]
if (st==s):
    print (s, " is Palindrome ")
else :
    print (s, " is Not Palindrome ")
```

Output:

Enter Any String- madam

madam is Palindrome

>>> ===== RESTART

=====

>>>

Enter Any String raman

namar is Not Palindrome

Workshop Assignment

1. Given two integers x and n , compute x^n .

Solution:

```
x=int(input("Enter the value of x"))
n=int(input("Enter the value of n"))
p=x**n
print("pow(x,n)=",p)
```

2. Compute the greatest common divisor and the least common multiple of two integers.

Greatest Common Divisor (GCD) or Highest Common Factor (HCF) of two positive integers is the largest positive integer that divides both numbers without remainder. It is useful for reducing fractions to be in its lowest terms.

Lowest Common Multiple (LCM) of two integers is the smallest integer that is a multiple of both numbers.

Solution:

```
# Python program to find the L.C.M. of two input number
# define gcd function
def gcd(x, y):
    """This function implements the Euclidian algorithm
    to find G.C.D. of two numbers"""
    while(y):
        x, y = y, x % y
```

```

print("The G.C.D. of", num1,"and", num2,"is ", x)

return x

# define lcm function
def lcm(x, y):

    """This function takes two
    integers and returns the L.C.M."""

    lcm = (x*y)//gcd(x,y)

    return lcm

num1 = int(input("Enter first number: "))
num2 = int(input("Enter second number: "))

print("The L.C.M. of", num1,"and", num2,"is", lcm(num1, num2))

```

3. Test if a number is equal to the sum of the cubes of its digits. Find the smallest and largest such numbers in the range of 100 to 1000.

An “Armstrong number” is a number that is equal to the sum of the nth powers of its individual digits. For example, 371 is an Armstrong number where it has 3 digits and $3^3+7^3+1^3 = 371$

Python Program to Check Armstrong Number

```

num = int(input("enter a number: "))

length = len(str(num))
sum = 0
temp = num

while(temp != 0):
    sum = sum + ((temp % 10) ** length)
    temp = temp // 10

if sum == num:

```

```
        print("armstrong number")
else:
    print("not armstrong number")
```

Solution:

```
lower = 100
upper = 1000
l=[]
for num in range(lower, upper + 1):

    # order of number
    order = len(str(num))

    # initialize sum
    sum = 0

    # find the sum of the cube of each digit
    temp = num
    while temp > 0:
        digit = temp % 10
        sum += digit ** order
        temp //= 10

    if num == sum:
        l.append(num)

print("Smallest number=",l[0])
print("Largest number=",l[-1])
```

Program to find simple interest.

Program/Source Code

Here is source code of the Python Program to compute simple interest given all the required values. The program output is also shown below.

```
principle=float(input("Enter the principle amount:"))
time=int(input("Enter the time(years):"))
rate=float(input("Enter the rate:"))
simple_interest=(principle*time*rate)/100
print("The simple interest is:",simple_interest)
```

Program Explanation

1. User must enter the values for the principle amount, rate and time.
2. The formula: $(\text{amount} \times \text{time} \times \text{rate}) / 100$ is used to compute simple interest.
3. The simple interest is later printed.

Runtime Test Cases

Case 1:

Enter the principle amount:200

Enter the time(years):5

Enter the rate:5.0

The simple interest is: 50.0

Case 2:

Enter the principle amount:70000

Enter the time(years):1

Enter the rate:4.0

The simple interest is: 2800.0

Program to calculate Standard deviation.

```
from math import sqrt

def standard_deviation(lst, population=True):
    """Calculates the standard deviation for a list of numbers."""
    num_items = len(lst)
    mean = sum(lst) / num_items
    differences = [x - mean for x in lst]
    sq_differences = [d ** 2 for d in differences]
    ssd = sum(sq_differences)

    # Note: it would be better to return a value and then print it
    outside

    # the function, but this is just a quick way to print out the values
    along

    # the way.

    if population is True:
        print("This is POPULATION standard deviation.")
        variance = ssd / num_items
    else:
        print("This is SAMPLE standard deviation.")
        variance = ssd / (num_items - 1)
    sd = sqrt(variance)

    # You could `return sd` here.
```

```
print("The mean of {} is {}".format(lst, mean))
print("The differences are {}".format(differences))
print("The sum of squared differences is {}".format(ssd))
print("The variance is {}".format(variance))
print("The standard deviation is {}".format(sd))
print('-----')
```

```
s = [98, 127, 133, 147, 170, 197, 201, 211, 255]
standard_deviation(s)
standard_deviation(s, population=False)
```

Output:

This is POPULATION standard deviation.

The mean of [98, 127, 133, 147, 170, 197, 201, 211, 255] is 171.0.

The differences are [-73.0, -44.0, -38.0, -24.0, -1.0, 26.0, 30.0, 40.0, 84.0].

The sum of squared differences is 19518.0.

The variance is 2168.6666666666665.

The standard deviation is 46.56894530335282.

This is SAMPLE standard deviation.

The mean of [98, 127, 133, 147, 170, 197, 201, 211, 255] is 171.0.

The differences are [-73.0, -44.0, -38.0, -24.0, -1.0, 26.0, 30.0, 40.0, 84.0].

The sum of squared differences is 19518.0.

The variance is 2439.75.

The standard deviation is 49.393825525059306.

Program to find correlation coefficient

Given two array elements and we have to find the correlation coefficient between two array. Correlation coefficient is an equation that is used to determine the strength of relation between two variables. Correlation coefficient sometimes called as cross correlation coefficient. Correlation coefficient always lies between -1 to +1 where -1 represents X and Y are negatively correlated and +1 represents X and Y are positively correlated.

$$r = \frac{n(\sum xy) - (\sum x)(\sum y)}{\sqrt{[n\sum x^2 - (\sum x)^2][n\sum y^2 - (\sum y)^2]}}$$

Where r is correlation coefficient.

X	Y	X*Y	X*X	Y*Y
15	25	375	225	625
18	25	450	324	625
21	27	567	441	729
24	31	744	576	961
27	32	864	729	1024
$\sum X=105$	$\sum Y=140$	$\sum X*Y=3000$	$\sum X*X=2295$	$\sum Y*Y=3964$

Correlation coefficient

$$= (5 * 3000 - 105 * 140)$$

$$\begin{aligned} & / \sqrt{(5 * 2295 - 105^2) * (5 * 3964 - 140^2)} \\ & = 300 / \sqrt{450 * 220} = 0.953463 \end{aligned}$$

Examples :

Input : X[] = {43, 21, 25, 42, 57, 59}

Y[] = {99, 65, 79, 75, 87, 81}

Output : 0.529809

Input : X[] = {15, 18, 21, 24, 27};

Y[] = {25, 25, 27, 31, 32}

Output : 0.953463

```
# Python Program to find correlation coefficient.  
import math
```

```
# function that returns correlation coefficient.
```

```
def correlationCoefficient(X, Y, n) :
```

```
    sum_X = 0
```

```
    sum_Y = 0
```

```
    sum_XY = 0
```

```
    squareSum_X = 0
```

```
    squareSum_Y = 0
```

```
    i = 0
```

```
    while i < n :
```

```
        # sum of elements of array X.
```

```
        sum_X = sum_X + X[i]
```

```
        # sum of elements of array Y.
```

```
        sum_Y = sum_Y + Y[i]
```



```

# sum of X[i] * Y[i].
sum_XY = sum_XY + X[i] * Y[i]

# sum of square of array elements.
squareSum_X = squareSum_X + X[i] * X[i]
squareSum_Y = squareSum_Y + Y[i] * Y[i]

i = i + 1

# use formula for calculating correlation
# coefficient.
corr = (float)(n * sum_XY - sum_X * sum_Y)/
      (float)(math.sqrt((n * squareSum_X -
        sum_X * sum_X)* (n * squareSum_Y -
        sum_Y * sum_Y)))
return corr

# Driver function
X = [15, 18, 21, 24, 27]
Y = [25, 25, 27, 31, 32]

# Find the size of array.
n = len(X)

# Function call to correlationCoefficient.
print ('{0:.6f}'.format(correlationCoefficient(X, Y, n)))

```

Program for EMI Calculator

EMI stand for Equated Monthly Installment. This calculator is used to calculate per month EMI of loan amount if loan amount that is principal, rate of interest and time in years is given as input.

Formula:

$$E = \frac{P.r.(1+r)^n}{((1+r)^n - 1)}$$

Here,

P = loan amount i.e principal amount

$R = \text{Interest rate per month}$
 $T = \text{Loan time period in year}$

EMI Calculator program in Python

```
def emi_calculator(p, r, t):  
    r = r / (12 * 100) # one month interest  
    t = t * 12 # one month period  
    emi = (p * r * pow(1 + r, t)) / (pow(1 + r, t) - 1)  
    return emi
```

driver code

```
principal = 10000;  
rate = 10;  
time = 2;  
emi = emi_calculator(principal, rate, time);  
print("Monthly EMI is= ", emi)
```

Program to calculate GST from original and net prices

Given Original cost and Net price then calculate the percentage of GST

Examples:

Input : Netprice = 120, original_cost = 100

Output : GST = 20%

Input : Netprice = 105, original_cost = 100

Output : GST = 5%

How to calculate GST
GST (Goods and Services Tax) which is included in netprice of product for get GST % first need to calculate GST Amount by subtract original cost from Netprice and then apply GST % formula = **(GST_Amount*100) / original_cost**

$$\text{Netprice} = \text{original_cost} + \text{GST_Amount}$$

$$\text{GST_Amount} = \text{Netprice} - \text{original_cost}$$

$$\text{GST_Percentage} = (\text{GST_Amount} * 100) / \text{original_cost}$$

```
# Python3 Program to
# compute GST from original
# and net prices.
```

```
def Calculate_GST(org_cost, N_price):

    # return value after calculate GST%
    return (((N_price - org_cost) * 100) / org_cost);

# Driver program to test above functions
org_cost = 100
N_price = 120
print("GST = ",end=")

print(round(Calculate_GST(org_cost, N_price)),end=")

print("%")
```

Linear Search

In computer science, **linear search** or **sequential search** is a method for finding a target value within a list. It sequentially checks each element of the list for the target value until a match is found or until all the elements have been searched.

Algorithm

Linear search sequentially checks each element of the list until it finds an element that matches the target value. If the algorithm reaches the end of the list, the search terminates unsuccessful.

Linear search is implemented using following steps...

Step 1: Read the search element from the user

Step 2: Compare, the search element with the first element in the list.

Step 3: If both are matching, then display "Given element found!!!" and terminate the function

Step 4: If both are not matching, then compare search element with the next element in the list.

Step 5: Repeat steps 3 and 4 until the search element is compared with the last element in the list.

Step 6: If the last element in the list is also doesn't match, then display "Element not found!!!" and terminate the function.

Example

Consider the following list of element and search element...

list

0	1	2	3	4	5	6	7
65	20	10	55	32	12	50	99

12

Both are not matching. So move to next element

Step 2:

search element (12) is compared with next element (20)

list

0	1	2	3	4	5	6	7
65	20	10	55	32	12	50	99

12

Both are not matching. So move to next element

Step 3:

search element (12) is compared with next element (10)

list

0	1	2	3	4	5	6	7
65	20	10	55	32	12	50	99

12

Both are not matching. So move to next element

Step 4:

search element (12) is compared with next element (55)

list

0	1	2	3	4	5	6	7
65	20	10	55	32	12	50	99

12

Both are not matching. So move to next element

Step 5:

search element (12) is compared with next element (32)

list

0	1	2	3	4	5	6	7
65	20	10	55	32	12	50	99

12

Both are not matching. So move to next element

Step 6:

search element (12) is compared with next element (12)

list

0	1	2	3	4	5	6	7
65	20	10	55	32	12	50	99

12

Both are matching. So we stop comparing and display element found at index 5.

Application

Linear search is usually very simple to implement, and is practical when the list has only a few elements, or when performing a single search in an unordered list

When many values have to be searched in the same list, it often pays to pre-process the list in order to use a faster method. For example, one may sort the list and use binary search, or build an efficient search data structure from it. Should the content of the list change frequently, repeated re-organization may be more trouble than it is worth.

As a result, even though in theory other search algorithms may be faster than linear search (for instance binary search), in practice even on medium-sized arrays (around 100 items or less) it might be infeasible to use anything else. On larger arrays, it only makes sense to use other, faster search methods if the data is large enough, because the initial time to prepare (sort) the data is comparable to many linear searches.

Program Code

```
#Linear Search
list_of_elements = [4, 2, 8, 9, 3, 7]

x = int(input("Enter number to search: "))

found = False

for i in range(len(list_of_elements)):
    if(list_of_elements[i] == x):
        found = True
```

```
print("%d found at %dth position"%(x,i))
break
```

```
if(found == False):
    print("%d is not in list"%x)
```

```
#To find the frequency of numbers in a list
my_list=[5,5,5,5,2,2,2,3,3,3,6,6,7,8,9,11]
print('Original List : ',my_list)
my_set=set(my_list)
d=dict.fromkeys(my_set,0)
print('Dictionary with 0 values :')
print(d)
for n in my_list:
    d[n]=d[n]+1
print('Element : Frequency')
print(d)
```

Introduction to Python Module:

A module is a logical organization of Python code. Related code are grouped into a module which makes the code easier to understand and use. Any python module is an object with different attributes

which can be bind and referenced. Simply, it is a file containing a set of functions which can be included in our application.

A module is a file containing Python definitions and statements. The file name is the module name with the suffix `.py` appended. Within a module, the module's name (as a string) is available as the value of the global variable `__name__`. For instance, use text editor to create a file called `fibonacci.py` in the current directory with the following contents:

```
# Fibonacci numbers module

def fib(n): # write Fibonacci series up to n
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()

def fib2(n): # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)
        a, b = b, a+b
    return result
```

Creating a Module:

It is very easy to create a module in python. we have to just save the code which we want to be as a part of module in a file with the file extension `.py` e.g.

1. Create a file a.py with following code

```
def label(str1):  
    print("-----")  
    print(str1)  
    print("-----")
```

2. Save this file

3. Now create one another file b.py with following code

```
import a  
  
s=input("enter your school name")  
  
a.label(s)
```

4. Now run it.

In above e.g. file a.py is being used as module b.py as main program. The module can contain not only functions but also variables of all types (arrays, dictionaries, objects etc)

Import modules in Python

Import in python is similar to #include header_file in C/C++. Python modules can get access to code from another module by importing the file/function using import. The import statement is the most common way of invoking the import machinery, but it is not the only way. Modules can be imported as:

i) import module_name

When import is used, it searches for the module initially in the local scope by calling `__import__()` function. The value returned by the function are then reflected in the output of the initial code. In the above (Fibonacci number module) example, if we write

Import fibo

This does not enter the names of the functions defined in `fib` directly in the current symbol table; it only enters the module name `fib` there. Using the module name you can access the functions:

```
>>>
>>> fibo.fib(1000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

If you intend to use a function often you can assign it to a local name:

```
>>>
>>> fib = fibo.fib
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

ii) **from module_name import function_names**

There is a variant of the [import](#) statement that imports names from a module directly into the importing module's symbol table. For example:

```
>>>
>>> from fibo import fib, fib2
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This does not introduce the module name from which the imports are taken in the local symbol table (so in the example, fibo is not defined).

iii) **from module_name import ***

There is even a variant to import all names that a module defines:

```
>>> from fibo import *
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This imports all names except those beginning with an underscore (_). In most cases Python programmers do not use this facility since it introduces an unknown set of names into the interpreter, possibly hiding some things you have already defined.

Re-naming a Module :We can alias a module while importing it with the help of as keyword

```
e.g.   import a as m1                #here a is module and
m1 is alias of the module
        s = m1.person1["age"]
        print(s)
```

Checking for built in module

1. Type import on shell. E.g. import math
2. If it shows error then it indicate that this module is not installed.
3. Install the module after download with the help of pip/pip3 command

Some built in modules Numeric and Mathematical Modules

1. numbers — Numeric abstract base classes
 2. math — Mathematical functions
 3. cmath — Mathematical functions for complex number
 4. decimal — Decimal fixed point and floating point arithmetic
 5. fractions — Rational numbers
 6. random — Generate pseudo-random numbers
 7. statistics — Mathematical statistics functions
- Functional Programming Modules
1. itertools — Functions creating iterators for efficient looping
 2. functools — Higher-order functions and operations on callable objects
 3. operator — Standard operators as functions

INTRODUCTION TO DATA STRUCTURES IN PANDAS

SERIES:

Series is a one-dimensional labeled array capable of holding data of any type (integer, string, float, python objects, etc.). The axis labels are collectively called index.

A pandas Series can be created using the following constructor –

```
pandas.Series( data, index, dtype, copy)
```

The parameters of the constructor are as follows –

S. No	Parameter & Description
------------------	------------------------------------

1	data data takes various forms like ndarray, list, constants
2	index Index values must be unique and hashable, same length as data. Default np.arange(n) if no index is passed.
3	dtype dtype is for data type. If None, data type will be inferred
4	copy Copy data. Default False

A series can be created using various inputs like –

- Array
- Dict
- Scalar value or constant

From within the interpreter, import both the numpy and pandas packages into your namespace:

```
In [1]: import numpy as np
```

```
In [2]: import pandas as pd
```

The basic method to create a Series is to call:

```
>>> s = pd.Series([data], index=[index])
```

We'll input integer data and then provide a name parameter for the Series, but we'll avoid using the index parameter to see how pandas populates it implicitly:

```
>>> s = pd.Series([0, 1, 4, 9, 16, 25], name='Squares')
```

Now, let's call the Series so we can see what pandas does with it:

```
>>> s
```

We'll see the following output, with the index in the left column, our data values in the right column. Below the columns is information about the Name of the Series and the data type that makes up the values.

Output

```
0    0
1    1
2    4
3    9
4   16
5   25
```

Name: Squares, dtype: int64

Though we did not provide an index for the array, there was one added implicitly of the integer values 0 through 5.

Declaring an Index

As the syntax above shows us, we can also make Series with an explicit index. We'll use data about the average depth in meters of the Earth's oceans:

```
avg_ocean_depth = pd.Series([1205, 3646, 3741, 4080, 3270],
index=['Arctic', 'Atlantic', 'Indian', 'Pacific', 'Southern'])
```

With the Series constructed, let's call it to see the output:

```
>>> avg_ocean_depth
```

Output

```
Arctic    1205
Atlantic  3646
Indian    3741
Pacific   4080
Southern  3270
dtype: int64
```

We can see that the index we provided is on the left with the values on the right.

Indexing and Slicing Series

With pandas Series we can index by corresponding number to retrieve values:

```
>>> avg_ocean_depth[2]
```

WILL PRODUCE OUTPUT AS

Output

```
3741
```

We can also slice by index number to retrieve values:

```
>>> avg_ocean_depth[2:4]
```

WILL PRODUCE OUTPUT AS

Output

```
Indian    3741
Pacific   4080
dtype: int64
```

Additionally, we can call the value of the index to return the value that it corresponds with:

```
>>> avg_ocean_depth['Indian']
```

Will produce output as

Output
3741

Create a Series from dictionary

A **dict** can be passed as input and if no index is specified, then the dictionary keys are taken in a sorted order to construct index. If **index** is passed, the values in data corresponding to the labels in the index will be pulled out.

Example 1

```
#import the pandas library and aliasing as pd
import pandas as pd
import numpy as np
data = {'a' : 0., 'b' : 1., 'c' : 2.}
s = pd.Series(data)
print s
```

Its **output** is as follows –

```
a 0.0
b 1.0
c 2.0
dtype: float64
```

Example 2

```
#import the pandas library and aliasing as pd
import pandas as pd
```



```
import numpy as np
data = {'a' : 0., 'b' : 1., 'c' : 2.}
s = pd.Series(data,index=['b','c','d','a'])
print s
```

Its **output** is as follows –

```
b 1.0
c 2.0
d NaN
a 0.0
dtype: float64
```

Observe – Index order is persisted and the missing element is filled with NaN (Not a Number).

Create a Series from Scalar

If data is a scalar value, an index must be provided. The value will be repeated to match the length of **index**

```
#import the pandas library and aliasing as pd
import pandas as pd
import numpy as np
s = pd.Series(5, index=[0, 1, 2, 3])
print s
```

Its **output** is as follows –

```
0 5
1 5
2 5
3 5
dtype: int64
```

Accessing Data from Series with Position

Data in the series can be accessed similar to that in an **ndarray**.

Example 1

Retrieve the first element. As we already know, the counting starts from zero for the array, which means the first element is stored at zeroth position and so on.

```
import pandas as pd
s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])

#retrieve the first element
print s[0]
```

Its **output** is as follows –

```
1
```

Example 2

Retrieve the first three elements in the Series. If a `:` is inserted in front of it, all items from that index onwards will be extracted. If two parameters (with `:` between them) is used, items between the two indexes (not including the stop index)

```
import pandas as pd
s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])

#retrieve the first three element
print s[:3]
```

Its **output** is as follows –

```
a 1
b 2
c 3
dtype: int64
```

Example 3

Retrieve the last three elements.

```
import pandas as pd
s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])

#retrieve the last three element

print s[-3:]
```

Its **output** is as follows -

```
c 3
d 4
e 5
dtype: int64
```

Retrieve Data Using Label (Index)

A Series is like a fixed-size **dict** in that you can get and set values by index label.

Example 1

Retrieve a single element using index label value.

```
import pandas as pd
s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])

#retrieve a single element

print s['a']
```

Its **output** is as follows -

1

Example 2

Retrieve multiple elements using a list of index label values.

```
import pandas as pd
s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])

#retrieve multiple elements
print s[['a','c','d']]
```

Its **output** is as follows –

```
a 1
c 3
d 4
dtype: int64
```

QUESTIONS ON PANDS SERIES

1. What do you understand by Series?
2. Write a code to create an empty series.
3. Write a Python program to create and display a one-dimensional array-like object containing an array of data .

Sol: **Python Code :**

```
import pandas as pd
ds = pd.Series([2, 4, 6, 8, 10])
print(ds)
```

Sample Output:

```
0    2
1    4
2    6
3    8
4   10
dtype: int64
```

4. Write a Python program to convert a Panda module Series to Python list and it's type.

Python Code :

```
import pandas as pd

ds = pd.Series([2, 4, 6, 8, 10])

print("Pandas Series and type")

print(ds)

print(type(ds))

print("Convert Pandas Series to Python list")

print(ds.tolist())

print(type(ds.tolist()))
```

Sample Output:

```
Pandas Series and type
0    2
1    4
2    6
3    8
4   10
dtype: int64
<class 'pandas.core.series.Series'>
Convert Pandas Series to Python list
[2, 4, 6, 8, 10]
<class 'list'>
```

5. Write a Python program to get the largest integer smaller or equal to the division of the inputs.

```
import pandas as pd

ds1 = pd.Series([2, 4, 6, 8, 10])
ds2 = pd.Series([1, 3, 5, 7, 10])

print("Series1:")
print(ds1)

print("Series2:")
print(ds2)

print("Compare the elements of the said Series:")
print("Equals:")
print(ds1 == ds2)

print("Greater than:")
print(ds1 > ds2)

print("Less than:")
print(ds1 < ds2)
```

Sample Output:

```
1. Series1:
2. 0    2
3. 1    4
4. 2    6
5. 3    8
6. 4   10
7. dtype: int64
8. Series2:
9. 0    1
10. 1    3
```

11. 2 5
12. 3 7
13. 4 10
14. dtype: int64
15. Compare the elements of the said Series:
16. Equals:
17. 0 False
18. 1 False
19. 2 False
20. 3 False
21. 4 True
22. dtype: bool
23. Greater than:
24. 0 True
25. 1 True
26. 2 True
27. 3 True
28. 4 False
29. dtype: bool
30. Less than:
31. 0 False
32. 1 False
33. 2 False
34. 3 False
35. 4 False
36. dtype: bool

DATA FRAMES:

A Data frame is a two-dimensional data structure, i.e., data is aligned in a tabular fashion in rows and columns.

Features of DataFrame

- Potentially columns are of different types
- Size – Mutable
- Labeled axes (rows and columns)
- Can Perform Arithmetic operations on rows and columns
- Structure
- Let us assume that we are creating a data frame with student's data.

Regd. No	Name	Marks%
1000	Steve	86.29
1001	Mathew	91.63
1002	Jose	72.90
1003	Patty	69.23
1004	Vin	88.30

pandas.DataFrame

A pandas DataFrame can be created using the following constructor –

```
pandas.DataFrame( data, index, columns, dtype, copy)
```

The parameters of the constructor are as follows –

.N	Parameter & Description
1	<p>data</p> <p>data takes various forms like ndarray, series, map, lists, dict, constants and also another DataFrame.</p>

2	index For the row labels, the Index to be used for the resulting frame is Optional Default np.arrange(n) if no index is passed.
3	columns For column labels, the optional default syntax is - np.arrange(n). This is only true if no index is passed.
4	dtype Data type of each column.
4	copy This command (or whatever it is) is used for copying of data, if the default is False.

Create DataFrame

A pandas DataFrame can be created using various inputs like –

- Lists
- dict
- Series
- Numpy ndarrays
- Another DataFrame

In the subsequent sections of this chapter, we will see how to create a DataFrame using these inputs.

Create an Empty DataFrame

A basic DataFrame, which can be created is an Empty Dataframe.

Example

```
#import the pandas library and aliasing as pd
```

```
import pandas as pd
df = pd.DataFrame()
print df
```

Its **output** is as follows

```
Empty DataFrame
Columns: []
Index: []
```

Create a DataFrame from Lists

The DataFrame can be created using a single list or a list of lists.

Example 1

```
import pandas as pd
data = [1,2,3,4,5]
df = pd.DataFrame(data)
print df
```

Its **output** is as follows –

```
0
0  1
1  2
2  3
3  4
4  5
```

Example 2

```
import pandas as pd
data = [['Alex',10],['Bob',12],['Clarke',13]]
df = pd.DataFrame(data,columns=['Name','Age'])
print df
```

Its **output** is as follows –

	Name	Age
0	Alex	10
1	Bob	12
2	Clarke	13

QUESTIONS WITH ANSWERS ON DATA FRAME:

1. What is data frame.?
2. Write a code to create a data frame?
3. Write a Python program to get the powers of an array values element-wise.

Note: First array elements raised to powers from second array

Expected Output:

Original array

```
[0 1 2 3 4 5 6]
```

First array elements raised to powers from second array, element-wise:

```
[ 0 1 8 27 64 125 216]
```

```
import pandas as pd
```

```
df = pd.DataFrame({'X':[78,85,96,80,86],  
'Y':[84,94,89,83,86],'Z':[86,97,96,72,83]});
```

```
print(df)
```

Sample Output:

```
1.  X  Y  Z  
2. 0  78 84 86  
3. 1  85 94 97  
4. 2  96 89 96  
5. 3  80 83 72  
6. 4  86 86 83
```

4. Write a Python program to create and display a DataFrame from a specified dictionary data which has the index labels.

Python Code :

```
import pandas as pd
import numpy as np

exam_data = {'name': ['Anastasia', 'Dima', 'Katherine', 'James',
'Emily', 'Michael', 'Matthew', 'Laura', 'Kevin', 'Jonas'],
'score': [12.5, 9, 16.5, np.nan, 9, 20, 14.5, np.nan, 8, 19],
'attempts': [1, 3, 2, 3, 2, 3, 1, 1, 2, 1],
'qualify': ['yes', 'no', 'yes', 'no', 'no', 'yes', 'yes', 'no', 'no', 'yes']}

labels = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']

df = pd.DataFrame(exam_data , index=labels)

print(df)
```

Copy

Sample Output:

	attempts	name	qualify	score
a	1	Anastasia	yes	12.5
b	3	Dima	no	9.0
c	2	Katherine	yes	16.5
d	3	James	no	NaN
e	2	Emily	no	9.0
f	3	Michael	yes	20.0
g	1	Matthew	yes	14.5
h	1	Laura	no	NaN
i	2	Kevin	no	8.0
j	1	Jonas	yes	19.0

Operations on Series

A Series is a one-dimensional object that can hold any data type such as integers, floats and strings. Let's take a list of items as an input argument and create a Series object for that list.

```
>>> import pandas as pd
>>> x = pd.Series([6,3,4,6])
>>> x
0 6
1 3
2 4
3 6
dtype: int64
```

The axis labels for the data as referred to as the index. The length of index must be the same as the length of data. Since we have not passed any index in the code above, the default index will be created with values [0, 1, ... len(data) - 1]

Lets go ahead and define indexes for the data.

```
>>> x = pd.Series([6,3,4,6], index=['a', 'b', 'c', 'd'])
>>> x
a 6
b 3
c 4
d 6
dtype: int64
```

The index in left most column now refers to data in the right column.

We can lookup the data by referring to its index:

```
>>> x["c"]
```

4

Python gives us the relevant data for the index.

One example of a data type is the dictionary defined below. The index and values correlate to keys and values. We can use the index to get the values of data corresponding to the labels in the index.

```
>>> data = {'abc': 1, 'def': 2, 'xyz': 3}
>>> pd.Series(data)
abc 1
def 2
xyz 3
dtype: int64
```

Another interesting feature in Series is having data as a scalar value. In that case, the data value gets repeated for each of the indexes defined.

```
>>> x = pd.Series(3, index=['a', 'b', 'c', 'd'])
>>> x
a 3
b 3
c 3
d 3
dtype: int64
```

Series is a one-dimensional labeled array capable of holding data of any type (integer, string, float, python objects, etc.). The axis labels are collectively called index.

pandas.Series

A pandas Series can be created using the following constructor –

```
pandas.Series( data, index, dtype, copy)
```

The parameters of the constructor are as follows –

S.N	Parameter & Description
1	data data takes various forms like ndarray, list, constants
2	index Index values must be unique and hashable, same length as data. Default np.arange(n) if no index is passed.
3	dtype dtype is for data type. If None, data type will be inferred
4	copy Copy data. Default False

A series can be created using various inputs like –

- Array
- Dict
- Scalar value or constant

Create an Empty Series

A basic series, which can be created is an Empty Series.

Example

```
#import the pandas library and aliasing as pd
import pandas as pd
s = pd.Series()
print s
```


Its **output** is as follows –

```
Series([], dtype: float64)
```

Create a Series from ndarray

If data is an ndarray, then index passed must be of the same length. If no index is passed, then by default index will be **range(n)** where **n** is array length, i.e., **[0,1,2,3.... range(len(array))-1]**.

Example 1

```
#import the pandas library and aliasing as pd
import pandas as pd
import numpy as np
data = np.array(['a','b','c','d'])
s = pd.Series(data)
print s
```

Its **output** is as follows –

```
0 a
1 b
2 c
3 d
dtype: object
```

We did not pass any index, so by default, it assigned the indexes ranging from 0 to **len(data)-1**, i.e., 0 to 3.

Example 2

```
#import the pandas library and aliasing as pd
import pandas as pd
import numpy as np
data = np.array(['a','b','c','d'])
```

```
s = pd.Series(data,index=[100,101,102,103])  
print s
```

Its **output** is as follows –

```
100 a  
101 b  
102 c  
103 d  
dtype: object
```

We passed the index values here. Now we can see the customized indexed values in the output.

Create a Series from dict

A **dict** can be passed as input and if no index is specified, then the dictionary keys are taken in a sorted order to construct index. If **index** is passed, the values in data corresponding to the labels in the index will be pulled out.

Example 1

```
#import the pandas library and aliasing as pd  
import pandas as pd  
import numpy as np  
data = {'a' : 0., 'b' : 1., 'c' : 2.}  
s = pd.Series(data)  
print s
```

Its **output** is as follows –

```
a 0.0  
b 1.0  
c 2.0  
dtype: float64
```

Observe – Dictionary keys are used to construct index.

Example 2

```
#import the pandas library and aliasing as pd
import pandas as pd
import numpy as np
data = {'a' : 0., 'b' : 1., 'c' : 2.}
s = pd.Series(data,index=['b','c','d','a'])
print s
```

Its **output** is as follows –

```
b 1.0
c 2.0
d NaN
a 0.0
dtype: float64
```

Observe – Index order is persisted and the missing element is filled with NaN (Not a Number).

Create a Series from Scalar

If data is a scalar value, an index must be provided. The value will be repeated to match the length of **index**

```
#import the pandas library and aliasing as pd
import pandas as pd
import numpy as np
s = pd.Series(5, index=[0, 1, 2, 3])
print s
```

Its **output** is as follows –

```
0 5
1 5
2 5
```

```
3 5
dtype: int64
```

Accessing Data from Series with Position

Data in the series can be accessed similar to that in an **ndarray**.

Example 1

Retrieve the first element. As we already know, the counting starts from zero for the array, which means the first element is stored at zeroth position and so on.

```
import pandas as pd
s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])

#retrieve the first element
print s[0]
```

Its **output** is as follows –

```
1
```

Example 2

Retrieve the first three elements in the Series. If a `:` is inserted in front of it, all items from that index onwards will be extracted. If two parameters (with `:` between them) is used, items between the two indexes (not including the stop index)

```
import pandas as pd
s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])

#retrieve the first three element
print s[:3]
```

Its **output** is as follows –

```
a 1
b 2
c 3
dtype: int64
```

Example 3

Retrieve the last three elements.

```
import pandas as pd
s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])

#retrieve the last three element

print s[-3:]
```

Its **output** is as follows –

```
c 3
d 4
e 5
dtype: int64
```

Retrieve Data Using Label (Index)

A Series is like a fixed-size **dict** in that you can get and set values by index label.

Example 1

Retrieve a single element using index label value.

```
import pandas as pd
s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])

#retrieve a single element

print s['a']
```

Its **output** is as follows –

1

Example 2

Retrieve multiple elements using a list of index label values.

```
import pandas as pd
s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])

#retrieve multiple elements
print s[['a','c','d']]
```

Its **output** is as follows –

```
a 1
c 3
d 4
dtype: int64
```

Example 3

If a label is not contained, an exception is raised.

```
import pandas as pd
s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])

#retrieve multiple elements
print s['f']
```

Its **output** is as follows –

```
...
KeyError: 'f'
```

Questions:-

1. Define Series with an Example.
2. What is the use of keyword “import”?

Pandas DataFrame operations

(Create, display, iteration, select column, add column, delete column)

Prepared by: Pravin Singh, PGT (CS), KVS(RO), Ranchi
Excerpt from: Python for Data Analysis by Wes McKinney

Prerequisite

Basic understanding of common data structure of Python like list, tuple, dict, etc. is the prerequisite of Pandas.

The Basics:

To work with pandas in Python, the very first instruction should be import directive which may be done as follow.

In [1]: import pandas as pd

Or may be more specific

In [2]: from pandas import DataFrame

To get started with pandas, you will need to get comfortable with its two workhorse data structures: **Series and DataFrame**.

DataFrame:

A DataFrame represents a rectangular table of data and contains an ordered collection of columns, each of which can be a different value type (numeric, string, boolean, etc.). The DataFrame has both a row and column index; it can be thought of as a dict of Series all sharing the same index. Under the hood, the data is stored as one or more two-dimensional blocks rather than a list, dict, or some other collection of one-dimensional arrays.

While a DataFrame is physically two-dimensional, you can use it to represent higher dimensional data in a tabular format using hierarchical indexing.

Creating DataFrame and Displaying DataFrame:

There are many ways to construct a DataFrame, though one of the most common is from a dict of equal-length lists or NumPy arrays:

```
data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada', 'Nevada'],
'year': [2000, 2001, 2002, 2001, 2002, 2003],
'pop': [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}
frame = pd.DataFrame(data)
```

The resulting DataFrame will have its index assigned automatically as with Series, and the columns are placed in sorted order:

```
In [45]: frame
```

```
Out[45]:
```

```
      pop      state      year
```


0	1.5	Ohio	2000
1	1.7	Ohio	2001
2	3.6	Ohio	2002
3	2.4	Nevada	2001
4	2.9	Nevada	2002
5	3.2	Nevada	2003

For large DataFrames, the head method selects only the first five rows:

In [45]: frame

Out[45]:

	pop	state	year
0	1.5	Ohio	2000
1	1.7	Ohio	2001
2	3.6	Ohio	2002
3	2.4	Nevada	2001
4	2.9	Nevada	2002

If you specify a sequence of columns, the DataFrame's columns will be arranged in that order:

In [47]: pd.DataFrame(data, columns=['year', 'state', 'pop'])

Out[47]:

	year	state	pop
0	2000	Ohio	1.5
1	2001	Ohio	1.7
2	2002	Ohio	3.6
3	2001	Nevada	2.4

```
4    2002    Nevada    2.9
5    2003    Nevada    3.2
```

A column in a DataFrame can be retrieved as a Series either by dict-like notation or by attribute:

```
In [51]: frame2['state']
```

```
Out[51]:
```

```
one    Ohio
two    Ohio
three  Ohio
four   Nevada
five   Nevada
six    Nevada
```

```
Name: state, dtype: object
```

```
In [52]: frame2.year
```

```
Out[52]:
```

```
one    2000
two    2001
three  2002
four   2001
five   2002
six    2003
```

```
Name: year, dtype: int64
```

`frame2[column]` works for any column name, but `frame2.column` only works when the column name is a valid Python variable name.

Rows can also be retrieved by position or name with the special `loc` attribute (much more on this later):

```
In [53]: frame2.loc['three']
```

```
Out[53]:
```

```
year    2002
```

```
state   Ohio
```

```
pop     3.6
```

```
debt    NaN
```

```
Name: three, dtype: object
```

Adding or Modifying DataFrame Column:

Columns can be added or modified by assignment. For example, the empty 'debt' column could be assigned a scalar value or an array of values:

```
In [54]: frame2['debt'] = 16.5
```

```
In [55]: frame2
```

```
Out[55]:
```

	year	state	pop	debt
one	2000	Ohio	1.5	16.5
two	2001	Ohio	1.7	16.5
three	2002	Ohio	3.6	16.5
four	2001	Nevada	2.4	16.5
five	2002	Nevada	2.9	16.5
six	2003	Nevada	3.2	16.5

When you are assigning lists or arrays to a column, the value's length must match the length of the DataFrame. If you assign a Series, its labels will be realigned exactly to the DataFrame's index, inserting missing values in any holes:

```
In [58]: val = pd.Series([-1.2, -1.5, -1.7], index=['two', 'four', 'five'])
```

```
In [59]: frame2['debt'] = val
```

```
In [60]: frame2
```

```
Out[60]:
```

	year	state	pop	debt
one	2000	Ohio	1.5	NaN
two	2001	Ohio	1.7	-1.2
three	2002	Ohio	3.6	NaN
four	2001	Nevada	2.4	-1.5
five	2002	Nevada	2.9	-1.7
six	2003	Nevada	3.2	NaN

Assigning a column that doesn't exist will create a new column.

Deleting DataFrame Column

The `del` keyword will delete columns as with a dict. As an example of `del`, I first add a new column of boolean values where the state column equals 'Ohio':

```
In [61]: frame2['eastern'] = frame2.state == 'Ohio'
```

```
In [62]: frame2
```

```
Out[62]:
```

	year	state	pop	debt	eastern
one	2000	Ohio	1.5	NaN	True
two	2001	Ohio	1.7	-1.2	True
three	2002	Ohio	3.6	NaN	True
four	2001	Nevada	2.4	-1.5	False

```
five 2002 Nevada 2.9 -1.7 False
```

```
six 2003 Nevada 3.2 NaN False
```

The del method can then be used to remove this column:

```
In [63]: del frame2['eastern']
```

```
In [64]: frame2.columns
```

```
Out[64]: Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

Binary Operations in a Data Frame: add, sub, mul, div, radd, rsub

DataFrame has the methods `add()`, `sub()`, `mul()`, `div()` and related functions `radd()`, `rsub()`, . . . for carrying out binary operations. For broadcasting behavior, Series input is of primary interest. Using these functions, you can use to either match on the index or columns via the axis keyword:

```
In [14]: df = pd.DataFrame({'one' : pd.Series(np.random.randn(3),  
.....:                                     'two' : pd.Series(np.random.randn(4), index=['a',  
'b', 'c', 'd']),  
.....:                                     'three' : pd.Series(np.random.randn(3),  
index=['b', 'c', 'd'])})  
.....:
```

```
In [15]: df
```

```
Out[15]:
```

```
      one      two      three  
a -1.101558  1.124472      NaN  
b -0.177289  2.487104 -0.634293  
c  0.462215 -0.486066  1.931194  
d      NaN -0.456288 -1.222918
```

```
In [16]: row = df.iloc[1]
```

In [17]: column = df['two']

In [18]: df.sub(row, axis='columns')

Out[18]:

```
      one    two    three
a -0.924269 -1.362632    NaN
b  0.000000  0.000000  0.000000
c  0.639504 -2.973170  2.565487
d     NaN -2.943392 -0.588625
```

In [19]: df.sub(row, axis=1)

Out[19]:

```
      one    two    three
a -0.924269 -1.362632    NaN
b  0.000000  0.000000  0.000000
c  0.639504 -2.973170  2.565487
d     NaN -2.943392 -0.588625
```

In [20]: df.sub(column, axis='index')

Out[20]:

```
      one two    three
a -2.226031  0.0    NaN
b -2.664393  0.0 -3.121397
c  0.948280  0.0  2.417260
d     NaN  0.0 -0.766631
```

In [21]: df.sub(column, axis=0)

Out[21]:

```
      one two    three
a -2.226031  0.0    NaN
b -2.664393  0.0 -3.121397
c  0.948280  0.0  2.417260
d     NaN  0.0 -0.766631
```

pandas.DataFrame.add

`DataFrame.add(other, axis='columns', level=None, fill_value=None)[source]`

Addition of dataframe and other, element-wise (binary operator *add*).

Equivalent to `dataframe + other`, but with support to substitute a fill value for missing data in one of the inputs.

Parameters:	<p>other : <i>Series, DataFrame, or constant</i></p> <p>axis : {0, 1, 'index', 'columns'} For Series input, axis to match Series index on</p> <p>level : <i>int or name</i> Broadcast across a level, matching Index values on the passed MultiIndex level</p> <p>fill_value : <i>None or float value, default None</i> Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing</p>
--------------------	--

Example

```
>>> a = pd.DataFrame([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'],
...                   columns=['one'])
>>> a
   one
a  1.0
b  1.0
c  1.0
d  NaN
>>> b = pd.DataFrame(dict(one=[1, np.nan, 1, np.nan],
...                         two=[np.nan, 2, np.nan, 2]),
...                   index=['a', 'b', 'd', 'e'])
>>> b
   one two
```

```

a 1.0 NaN
b NaN 2.0
d 1.0 NaN
e NaN 2.0
>>> a.add(b, fill_value=0)
   one two
a 2.0 NaN
b 1.0 2.0
c 1.0 NaN
d 1.0 NaN
e NaN 2.0

```

pandas.DataFrame.sub

DataFrame.**sub**(*other*, *axis*='columns', *level*=None, *fill_value*=None)[[source](#)]

Subtraction of dataframe and other, element-wise (binary operator *sub*).

Equivalent to dataframe - other, but with support to substitute a *fill_value* for missing data in one of the inputs.

Parameters:

other : Series, DataFrame, or constant

axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

level : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

fill_value : None or float value, default None

Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Example:


```

>>> a = pd.DataFrame([2, 1, 1, np.nan], index=['a', 'b', 'c', 'd'],
...                   columns=['one'])
>>> a
   one
a  2.0
b  1.0
c  1.0
d  NaN
>>> b = pd.DataFrame(dict(one=[1, np.nan, 1, np.nan],
...                       two=[3, 2, np.nan, 2]),
...                   index=['a', 'b', 'd', 'e'])
>>> b
   one two
a  1.0  3.0
b  NaN  2.0
d  1.0  NaN
e  NaN  2.0
>>> a.sub(b, fill_value=0)
   one two
a  1.0 -3.0
b  1.0 -2.0
c  1.0  NaN
d -1.0  NaN
e  NaN -2.0

```

pandas.DataFrame.mul

`DataFrame.mul`(*other*, *axis*='columns', *level*=None, *fill_value*=None)
 Multiplication of dataframe and other, element-wise (binary operator *mul*).

Equivalent to `dataframe * other`, but with support to substitute a `fill_value` for missing data in one of the inputs.

Parameters:

other : Series, DataFrame, or constant

axis : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

fill_value : *None or float value, default None*
Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing

level : *int or name*
Broadcast across a level, matching Index values on the passed MultiIndex level

Returns: **result** : *DataFrame*

Example:

pandas.DataFrame.div

DataFrame.**div**(*other, axis='columns', level=None, fill_value=None*)
Floating division of dataframe and other, element-wise (binary operator *truediv*).

Equivalent to `dataframe / other`, but with support to substitute a `fill_value` for missing data in one of the inputs.

Parameters:

other : *Series, DataFrame, or constant*
axis : {0, 1, 'index', 'columns'}
For Series input, axis to match Series index on

fill_value : *None or float value, default None*
Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing

level : *int or name*
Broadcast across a level, matching Index values on the passed MultiIndex level

Returns: **result** : *DataFrame*

Example:

pandas.DataFrame.radd

DataFrame.**radd**(*other*, axis='columns', level=None, fill_value=None)
Addition of dataframe and other, element-wise (binary operator *radd*).

Equivalent to `other + dataframe`, but with support to substitute a `fill_value` for missing data in one of the inputs.

Parameters:	other : Series, DataFrame, or constant
	axis : {0, 1, 'index', 'columns'} For Series input, axis to match Series index on
	fill_value : None or float value, default None Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing
	level : int or name Broadcast across a level, matching Index values on the passed MultiIndex level
Returns:	result : DataFrame

Example:

pandas.DataFrame.rsub

DataFrame.**rsub**(*other*, axis='columns', level=None, fill_value=None)
Subtraction of dataframe and other, element-wise (binary operator *rsub*).

Equivalent to `other - dataframe`, but with support to substitute a `fill_value` for missing data in one of the inputs.

Parameters:	other : Series, DataFrame, or constant
	axis : {0, 1, 'index', 'columns'} For Series input, axis to match Series index on
	fill_value : None or float value, default None

Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing

level : *int or name*

Broadcast across a level, matching Index values on the passed MultiIndex level

Returns: **result** : *DataFrame*

Example:

```
>>> a = pd.DataFrame([2, 1, 1, np.nan], index=['a', 'b', 'c', 'd'],
...                   columns=['one'])
>>> a
   one
a  2.0
b  1.0
c  1.0
d  NaN
>>> b = pd.DataFrame(dict(one=[1, np.nan, 1, np.nan],
...                       two=[3, 2, np.nan, 2]),
...                   index=['a', 'b', 'd', 'e'])
>>> b
   one two
a  1.0  3.0
b  NaN  2.0
d  1.0  NaN
e  NaN  2.0
>>> a.sub(b, fill_value=0)
   one two
a  1.0 -3.0
b  1.0 -2.0
c  1.0  NaN
d -1.0  NaN
e  NaN -2.0
```

pandas.DataFrame.rmul

DataFrame.**rmul**(*other*, axis='columns', level=None, fill_value=None)

Multiplication of dataframe and other, element-wise (binary operator *rmul*).

Equivalent to `other * dataframe`, but with support to substitute a `fill_value` for missing data in one of the inputs.

Parameters:	other : <i>Series, DataFrame, or constant</i>
	axis : {0, 1, 'index', 'columns'}
	For Series input, axis to match Series index on
	fill_value : <i>None or float value, default None</i>
	Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing
	level : <i>int or name</i>
	Broadcast across a level, matching Index values on the passed MultiIndex level
Returns:	result : <i>DataFrame</i>

Example:

pandas.DataFrame.rdiv

DataFrame.**rdiv**(*other*, axis='columns', level=None, fill_value=None)
Floating division of dataframe and other, element-wise (binary operator *rtruediv*).

Equivalent to `other / dataframe`, but with support to substitute a `fill_value` for missing data in one of the inputs.

Parameters:	other : <i>Series, DataFrame, or constant</i>
	axis : {0, 1, 'index', 'columns'}

Returns:	<p>For Series input, axis to match Series index on</p> <p>fill_value : <i>None or float value, default None</i> Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing</p> <p>level : <i>int or name</i> Broadcast across a level, matching Index values on the passed MultiIndex level</p> <p>result : <i>DataFrame</i></p>
Example:	

Operating on Data in Pandas

< [Data Indexing and Selection](#) | [Contents](#) | [Handling Missing Data](#) >

One of the essential pieces of NumPy is the ability to perform quick element-wise operations, both with basic arithmetic (addition, subtraction, multiplication, etc.) and with more sophisticated operations (trigonometric functions, exponential and logarithmic functions, etc.). Pandas inherits much of this functionality from NumPy, and the ufuncs that we introduced in [Computation on NumPy Arrays: Universal Functions](#) are key to this.

Pandas includes a couple useful twists, however: for unary operations like negation and trigonometric functions, these ufuncs will *preserve index and column labels* in the output, and for binary operations such as addition and multiplication, Pandas will automatically *align indices* when passing the objects to the ufunc. This means that keeping the context of data and combining data from different sources—both potentially error-prone tasks with raw

NumPy arrays—become essentially foolproof ones with Pandas. We will additionally see that there are well-defined operations between one-dimensional Series structures and two-dimensional DataFrame structures.

Ufuncs: Index Preservation

Because Pandas is designed to work with NumPy, any NumPy ufunc will work on Pandas Series and DataFrame objects. Let's start by defining a simple Series and DataFrame on which to demonstrate this:

```
import pandas as pd
import numpy as np
```

```
rng = np.random.RandomState(42)
ser = pd.Series(rng.randint(0, 10, 4))
ser
```

Output

```
0    6
1    3
2    7
3    4
```

```
dtype: int64
```

```
df = pd.DataFrame(rng.randint(0, 10, (3, 4)),
                  columns=['A', 'B', 'C', 'D'])
```

```
df
```

Output:

	A	B	C	D
0	6	9	2	6
1	7	4	3	7
2	7	2	5	4

If we apply a NumPy ufunc on either of these objects, the result will be another Pandas object *with the indices preserved*:

```
np.exp(x)
```

Output:

```
0    403.428793
1     20.085537
2   1096.633158
3     54.598150
dtype: float64
```

Or, for a slightly more complex calculation:

```
np.sin(x * np.pi / 4)
```

Output:

	A	B	C	D
0	1.000000	7.071068e-01	1.000000	1.000000e+00
1	0.707107	1.224647e-16	0.707107	-7.071068e-01
2	0.707107	1.000000e+00	0.707107	1.224647e-16

UFuncs: Index Alignment

For binary operations on two `Series` or `DataFrame` objects, Pandas will align indices in the process of performing the operation. This is very convenient when working with incomplete data, as we'll see in some of the examples that follow.

Index alignment in Series

As an example, suppose we are combining two different data sources, and find only the top three US states by *area* and the top three US states by *population*:

In [6]:

```
area = pd.Series({'Alaska': 1723337, 'Texas': 695662,  
                 'California': 423967}, name='area')  
population = pd.Series({'California': 38332521, 'Texas': 26448193,  
                       'New York': 19651127}, name='population')
```

Let's see what happens when we divide these to compute the population density:

```
population / area
```

Output:

```
Alaska      NaN  
California  90.413926  
New York    NaN  
Texas       38.018740  
dtype: float64
```

The resulting array contains the *union* of indices of the two input arrays, which could be determined using standard Python set arithmetic on these indices:

```
area.index | population.index
```

Output:

```
Index(['Alaska', 'California', 'New York', 'Texas'], dtype='object')
```

Any item for which one or the other does not have an entry is marked with NaN, or "Not a Number," which is how Pandas marks missing data (see further discussion of missing data in [Handling Missing Data](#)). This index matching is implemented this way for any of Python's built-in arithmetic expressions; any missing values are filled in with NaN by default:

```
A = pd.Series([2, 4, 6], index=[0, 1, 2])  
B = pd.Series([1, 3, 5], index=[1, 2, 3])
```

A + B

Output:

```
0 NaN
1 5.0
2 9.0
3 NaN
```

dtype: float64

If using NaN values is not the desired behavior, the fill value can be modified using appropriate object methods in place of the operators. For example, calling `A.add(B)` is equivalent to calling `A + B`, but allows optional explicit specification of the fill value for any elements in `A` or `B` that might be missing:

```
A.add(B, fill_value=0)
```

Output:

```
0 2.0
1 5.0
2 9.0
3 5.0
```

dtype: float64

Index alignment in DataFrame

A similar type of alignment takes place for *both* columns and indices when performing operations on DataFrames:

```
A = pd.DataFrame(rng.randint(0, 20, (2, 2)),
                  columns=list('AB'))
```

A

Output:

	A	B
0	1	11
1	5	1

```
B = pd.DataFrame(rng.randint(0, 10, (3, 3)),
                 columns=list('BAC'))
```

B

Output:

	B	A	C
0	4	0	9
1	5	8	0
2	9	2	6

In [13]:

```
A + B
```

Output:

	A	B	C
0	1.0	15.0	NaN
1	13.0	6.0	NaN
2	NaN	NaN	NaN

Notice that indices are aligned correctly irrespective of their order in the two objects, and indices in the result are sorted. As was the case with `Series`, we can use the associated object's arithmetic method and pass any desired `fill_value` to be used in place of missing entries. Here we'll fill with the mean of all values in A (computed by first stacking the rows of A):

In [14]:

```
fill = A.stack().mean()
A.add(B, fill_value=fill)
```

Output:

	A	B	C
--	----------	----------	----------

0	1.0	15.0	13.5
1	13.0	6.0	4.5
2	6.5	13.5	10.5

The following table lists Python operators and their equivalent Pandas object methods:

Python Operator	Pandas Method(s)
+	add()
-	sub(), subtract()
*	mul(), multiply()
/	truediv(), div(), divide()
//	floordiv()
%	mod()
**	pow()

Missing Data and Filling Values

When and Why Is Data Missed?

Let us consider an online survey for a product. Many a times, people do not share all the information related to them. Few people share their experience, but not how long they are using the product; few people share how long they are using the product, their experience but not their contact information. Thus, in some or the other way a part of data is always missing, and this is very common in real time.

Let us now see how we can handle missing values (say NA or NaN) using Pandas.

```

# import the pandas library

import pandas as pd

import numpy as np

df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e', 'f',
'h'], columns=['one', 'two', 'three'])

df = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])

print (df)

```

Its **output** is as follows –

	one	two	three
a	0.077988	0.476149	0.965836
b	NaN	NaN	NaN
c	-0.390208	-0.551605	-2.301950
d	NaN	NaN	NaN
e	-2.000303	-0.788201	1.510072
f	-0.930230	-0.670473	1.146615
g	NaN	NaN	NaN
h	0.085100	0.532791	0.887415

Using reindexing, we have created a DataFrame with missing values. In the output, **NaN** means **Not a Number**.

Check for Missing Values

To make detecting missing values easier (and across different array dtypes), Pandas provides the **isnull()** and **notnull()** functions, which are also methods on Series and DataFrame objects –

Example 1

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e', 'f',
'h'], columns=['one', 'two', 'three'])

df = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])

print (df['one'].isnull())
```

Its **output** is as follows –

```
a False
b True
c False
d True
e False
f False
g True
h False
Name: one, dtype: bool
```

Example 2

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e', 'f',
'h'], columns=['one', 'two', 'three'])

df = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])
```

```
print (df['one'].notnull())
```

Its **output** is as follows -

```
a True
b False
c True
d False
e True
f True
g False
h True
Name: one, dtype: bool
```

Calculations with Missing Data

- When summing data, NA will be treated as Zero
- If the data are all NA, then the result will be NA

Example 1

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e', 'f',
'h'], columns=['one', 'two', 'three'])

df = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])

print (df['one'].sum())
```

Its **output** is as follows –

```
2.02357685917
```

Example 2

```
import pandas as pd
import numpy as np
df = pd.DataFrame(index=[0,1,2,3,4,5],columns=['one','two'])
print (df['one'].sum())
```

Its **output** is as follows –

```
nan
```

Cleaning / Filling Missing Data

Pandas provides various methods for cleaning the missing values. The fillna function can “fill in” NA values with non-null data in a couple of ways, which we have illustrated in the following sections.

Replace NaN with a Scalar Value

The following program shows how you can replace "NaN" with "0".

```
import pandas as pd
import numpy as np
df = pd.DataFrame(np.random.randn(3, 3), index=['a', 'c',
'e'],columns=['one',
'two', 'three'])
df = df.reindex(['a', 'b', 'c'])
print df
```



```
print ("NaN replaced with '0':")
print (df.fillna(0))
```

Its **output** is as follows -

```
      one      two      three
a -0.576991 -0.741695  0.553172
b      NaN      NaN      NaN
c  0.744328 -1.735166  1.749580

NaN replaced with '0':
      one      two      three
a -0.576991 -0.741695  0.553172
b  0.000000  0.000000  0.000000
c  0.744328 -1.735166  1.749580
```

Here, we are filling with value zero; instead we can also fill with any other value.

Fill NA Forward and Backward

Using the concepts of filling discussed in the ReIndexing Chapter we will fill the missing values.

Method	Action
pad/fill	Fill methods Forward
bfill/backfill	Fill methods Backward

Example 1

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e', 'f,
```

```
'h'],columns=['one', 'two', 'three'])
df = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])

print (df.fillna(method='pad'))
```

Its **output** is as follows –

	one	two	three
a	0.077988	0.476149	0.965836
b	0.077988	0.476149	0.965836
c	-0.390208	-0.551605	-2.301950
d	-0.390208	-0.551605	-2.301950
e	-2.000303	-0.788201	1.510072
f	-0.930230	-0.670473	1.146615
g	-0.930230	-0.670473	1.146615
h	0.085100	0.532791	0.887415

Example 2

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e', 'f',
'h'],columns=['one', 'two', 'three'])
df = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])
print (df.fillna(method='backfill'))
```

Its **output** is as follows –

	one	two	three
a	0.077988	0.476149	0.965836
b	-0.390208	-0.551605	-2.301950
c	-0.390208	-0.551605	-2.301950

```
d -2.000303 -0.788201 1.510072
e -2.000303 -0.788201 1.510072
f -0.930230 -0.670473 1.146615
g 0.085100 0.532791 0.887415
h 0.085100 0.532791 0.887415
```

Series

Series is a one-dimensional labeled array capable of holding data of any type (integer, string, float, python objects, etc.). The axis labels are collectively called index.

A series can be created using various inputs like –

- Array
- Dict
- Scalar value or constant

A basic series, which can be created is an Empty Series.

Example

```
#import the pandas library and aliasing as pd
import pandas as pd
s = pd.Series()
print s
```

Output.
Series([], dtype: float64)

Create a Series from ndarray

If data is an ndarray, then index passed must be of the same length. If no index is passed, then by default index will be **range(n)** where **n** is array length, i.e., **[0,1,2,3.... range(len(array))-1]**.

Example 1

```
#import the pandas library and aliasing as pd
import pandas as pd
import numpy as np
data = np.array(['a','b','c','d'])
s = pd.Series(data)
print s
```

Output

```
0  a
1  b
2  c
3  d
dtype: object
```

Write a Python program to compare the elements of the two Pandas Series.

```
import pandas as pd
ds1 = pd.Series([2, 4, 6, 8, 10])
ds2 = pd.Series([1, 3, 5, 7, 10])
print("Series1:")
print(ds1)
print("Series2:")
print(ds2)
print("Compare the elements of the said Series:")
print("Equals:")
print(ds1 == ds2)
print("Greater than:")
print(ds1 > ds2)
print("Less than:")
print(ds1 < ds2)
```

Output

```
Series1:
0    2
1    4
2    6
3    8
```

```
4    10
dtype: int64
```

Series2:

```
0    1
1    3
2    5
3    7
4   10
dtype: int64
```

Compare the elements of the said Series:

Equals:

```
0    False
1    False
2    False
3    False
4     True
dtype: bool
```

Greater than:

```
0     True
1     True
2     True
3     True
4    False
dtype: bool
```

Less than:

```
0    False
1    False
2    False
3    False
4    False
dtype: bool
```

DataFrames


```
index = [2001, 2002, 2003, 2004])
```

```
df3 = pd.DataFrame({'HPI':[80,85,88,85],  
                    'Unemployment':[7, 8, 9, 6],  
                    'Low_tier_HPI':[50, 52, 50, 53]},  
                    index = [2001, 2002, 2003, 2004])
```

```
print(pd.merge(df1,df3, on='HPI'))
```

Output:

	HPI	Int_rate	US_GDP_Thousands	Low_tier_HPI	Unemployment
0	80	2	50	50	7
1	85	3	55	52	8
2	85	3	55	53	6
3	85	2	55	52	8
4	85	2	55	53	6
5	88	2	65	50	9

Loading CSV file to Dataframe Object

Create dataframe (that we will be importing)

```
import pandas as pd  
raw_data = {'first_name': ['Mohan', 'Abhijeet', 'Tina', 'Sanjay',  
                           'Sam'],  
            'last_name': ['Kumar', 'Singh', "Kumar", 'Dev', 'Singh'],  
            'age': [42, 52, 36, 24, 23],  
            'TestScore': [4, 24, 31, "23", "15"]}
```

```
df = pd.DataFrame(raw_data, columns = ['first_name',  
                                       'last_name', 'age', 'TestScore'])  
print(df)
```

	first_name	last_name	age	TestScore
0	Mohan	Kumar	42	4

1	Abhijeet	Singh	52	24
2	Tina	Kumar	36	31
3	Sanjay	Dev	24	23
4	Sam	Singh	23	15

Save dataframe as csv in the working directory

```
df.to_csv('example.csv')
```

Load a csv

```
df = pd.read_csv('example.csv')  
print(df)
```

	first_name	last_name	age	TestScore
0	Mohan	Kumar	42	4
1	Abhijeet	Singh	52	24
2	Tina	Kumar	36	31
3	Sanjay	Dev	24	23
4	Sam	Singh	23	15

Load a csv with no headers

```
df = pd.read_csv('example.csv', header=None)  
print(df)
```

	0	1	2	3
	first_name	last_name	age	TestScore
0	Mohan	Kumar	42	4

1	Abhijeet	Singh	52	24
2	Tina	Kumar	36	31
3	Sanjay	Dev	24	23
4	Sam	Singh	23	15

Load a csv while specifying column names

```
df = pd.read_csv('example.csv', names=['First Name', 'Last Name',  
'Age', 'Test Score'])  
print(df)
```

	First Name	Last Name	Age	Test Score
	first_name	last_name	age	TestScore
0	Mohan	Kumar	42	4
1	Abhijeet	Singh	52	24
2	Tina	Kumar	36	31
3	Sanjay	Dev	24	23
4	Sam	Singh	23	15

Firstly create table using following code:

```
import sqlite3

conn = sqlite3.connect('test.db')
print "Opened database successfully";

conn.execute("""CREATE TABLE COMPANY
              (ID INT PRIMARY KEY    NOT NULL,
              NAME          TEXT  NOT NULL,
              AGE           INT   NOT NULL,
              ADDRESS       CHAR(50),
              SALARY        REAL);""")
print ("Table created successfully");
```

Then Insert values to table using following:

```
conn.execute("INSERT INTO COMPANY
(ID,NAME,AGE,ADDRESS,SALARY) \
VALUES (1, 'Paul', 32, 'California', 20000.00 )");

conn.execute("INSERT INTO COMPANY
(ID,NAME,AGE,ADDRESS,SALARY) \
VALUES (2, 'Allen', 25, 'Texas', 15000.00 )");

conn.execute("INSERT INTO COMPANY
(ID,NAME,AGE,ADDRESS,SALARY) \
VALUES (3, 'Teddy', 23, 'Norway', 20000.00 )");

conn.execute("INSERT INTO COMPANY
(ID,NAME,AGE,ADDRESS,SALARY) \
VALUES (4, 'Mark', 25, 'Rich-Mond ', 65000.00 )");

conn.commit()
print ("Records created successfully");
```

Then you can import data to dataframe using the following:

```
import pandas as pd
```

```
df = pd.read_sql_query("SELECT id, name, address, salary from
COMPANY ",conn)
print(df)
conn.close()
```

You can also execute SQL query from Dataframe object as follow:

```
df.to_sql("UPDATE COMPANY set SALARY = 25000.00 where ID =
1",conn)
conn.commit()
df = pd.read_sql_query("SELECT id, name, address, salary from
COMPANY ",conn)
print(df)
conn.close()
```

To find largest and smallest number in a list.

Algorithm :

- Read input number asking for length of the list using `input()` or `raw_input()`.
- Initialise an empty list `lst = []`.
- Read each number using a for loop.
- In the for loop append each number to the list.
- Now we use predefined function `max()` to find the largest element in a list.
- Similarly we use another predefined function `min()` to find the smallest element in a list.

Program :

```
lst = []
num = int(input('How many numbers: '))
```

```
for n in range(num):
```

```
numbers = int(input('Enter number '))
lst.append(numbers)

print("Maximum element in the list is :", max(lst),
      "\nMinimum element in the list is :", min(lst))
```

Output :

```
How many numbers: 5
Enter number 10
Enter number 15
Enter number 5
Enter number 8
Enter number 30
Maximum number in a list : 30
Minimum number in a list : 5
```

Find the third largest number in a list

```
thelist = [1, 45, 88, 1, 45, 88, 5, 2, 103, 103, 7, 8]
theset = frozenset(thelist)
theset = sorted(theset, reverse=True)
print('1st = ' + str(theset[0]) + ' at ' + str(thelist.index(theset[0])))
print('2nd = ' + str(theset[1]) + ' at ' + str(thelist.index(theset[1])))
print('3rd = ' + str(theset[2]) + ' at ' + str(thelist.index(theset[2])))
```

Questions based on List

What is the result of this code?

```
nums = [5, 4, 3, 2, 1]
print(nums[1])
```

How many items are in this list?

```
[2,]
```

Which line of code will cause an error?

```
num = [5, 4, 3, [2], 1]
print(num[0])
print(num[3][0])
print(num[5])
```

What is the result of this code?

```
words = ["hello"]
words.append("world")
print(words[1])
```

What is the result of this code?

```
letters = ["a", "b", "c"]  
letters.append("d")  
print(len(letters))
```

What is the result of this code?

```
nums = [9, 8, 7, 6, 5]  
nums.append(4)  
nums.insert(2, 11)  
print(len(nums))
```

What is the result of this code?

```
letters = ["a", "b", "c"]  
letters.append("d")  
print(len(letters))
```

What is the result of this code?

```
nums = [9, 8, 7, 6, 5]  
nums.append(4)  
nums.insert(2, 11)  
print(len(nums))
```

What is the result of this code?

```
nums = list(range(5))  
print(nums[4])
```

What is the result of this code?

```
nums = list(range(5, 8))  
print(len(nums))
```

What is the result of this code?

```
nums = list(range(3, 15, 3))  
print(nums[2])
```

What is the result of this code?

```
sqs = [0, 1, 4, 9, 16, 25, 36, 49, 64]  
print(sqs[4:7])
```

What is the output of this code?

```
sqs = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]  
print(sqs[1::4])
```

What is the output of this code?

```
sqs = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
print(sqs[7:5:-1])
```

What does this list comprehension create?

```
nums = [i*2 for i in range(10)]
```

Create a list of multiples of 3 from 0 to 20.

PROGRAM#1: FIND THE SUM OF SQUARES OF THE FIRST 100 NATURAL

Program:

```
# Python3 Program to find sum of square of first n natural
numbers

# Return the sum of square of first n natural numbers def
squaresum(n) :
    # Iterate i from 1 and n finding square of i and add to sum.
    sm = 0
    for i in range(1, n+1) :
        sm = sm + (i * i)

    return sm

# Driven Program
n = 4
print(squaresum(n))
```



```
===== Method 2
=====
```

```
# Return the sum of square of first n natural numbers def
squaresum(n) :
```

```
return (n * (n + 1) * (2 * n + 1)) // 6
```

```
# Driven Program
```

```
n = 4
```

```
print(squaresum(n))
```

PROGRAM #2 : FIND WHETHER A STRING IS A PALINDROME OR NOT.

```
# Python3 code to find if given string is K-Palindrome or not
```

```
# Find if given string is K-Palindrome or not
```

```
def isKPalRec(str1, str2, m, n):
```

```
    # If first string is empty, the only option is to remove
```

```
    # all characters of second string if not m: return n
```

```
    # If second string is empty, the only option is to remove
```

```
    # all characters of first string if not n: return m
```

```
    if str1[m-1] == str2[n-1]:
```

```
        return isKPalRec(str1, str2, m-1, n-1)
```

```
    # If last characters are not same,
```

```
    # 1. Remove last char from str1 and recur for m-1 and n
```

```
    # 2. Remove last char from str2 and recur for m and n-1
```

```

# Take minimum of above two operations

res = 1 + min(isKPalRec(str1, str2, m-1, n), # Remove from str1
              (isKPalRec(str1, str2, m, n-1))) # Remove from str2

return res

# Returns true if str is k palindrome.

def isKPal(string, k):
    revStr = string[::-1]
    l = len(string)
    return (isKPalRec(string, revStr, 1, l) <= k * 2)
# Driver program
string = "acdcba"
k = 2

print("Yes" if isKPal(string, k) else "No")

```

```

===== END
=====

```

1) TO FIND THE POWER OF A NUMBER USING RECURSION.

Program Code:

```

def power(base,exp):
    if (exp==1):
        return (base)
    if (exp!=1):
        return (base*power(base,exp-1))
base = int(input("Enter base: "))
exp = int(input("Enter exponential value: "))
print ("Result:",power(base,exp))

```

Program Explanation

1. User must enter the base and exponential value.
2. The numbers are passed as arguments to a recursive function to find the power of the number.
3. The base condition is given that if the exponential power is equal to 1, the base number is returned.
4. If the exponential power isn't equal to 1, the base number multiplied with the power function is called recursively with the arguments as the base and power minus 1.
5. The final result is printed.

Runtime Test Cases

Case 1:

Enter base: 2

Enter exponential value: 5

Result: 32

Case 2:

Enter base: 5

Enter exponential value: 3

Result: 125

2) FUNCTION TO CALCULATE X RAISED TO THE POWER N:

```
def power(x, n):
```

```
    if (n == 0): return 1
```

```
    elif (int(n % 2) == 0):
```

```
        return (power(x, int(n / 2)) *  
                power(x, int(n / 2)))
```

```
else:
    return (x * power(x, int(n / 2)) *
            power(x, int(n / 2)))
```

```
# Driver Code
x = 2; y = 3
print(power(x, y))
```

3) TO COMPUTE GCD OF TWO INTEGERS:

```
# define a function
def computeGCD(x, y):

# choose the smaller number
    if x > y:
        smaller = y
    else:
        smaller = x
    for i in range(1, smaller+1):
        if((x % i == 0) and (y % i == 0)):
            hcf = i

    return hcf

num1 = 54
num2 = 24
# take input from the user
# num1 = int(input("Enter first number: "))
# num2 = int(input("Enter second number: "))

print("The H.C.F. of", num1,"and", num2,"is", computeGCD(num1,
num2))
```

Output:

The G.C.D. of 54 and 24 is 6

4) PROGRAM TO FIND THE L.C.M. OF TWO INTEGERS:

```

# define a function
def lcm(x, y):
    """This function takes two integers and returns the L.C.M."""
    # choose the greater number
    if x > y:
        greater = x
    else:
        greater = y

    while(True):
        if((greater % x == 0) and (greater % y == 0)):
            lcm = greater
            break
        greater += 1

    return lcm

# change the values of num1 and num2 for a different result
num1 = 54
num2 = 24

# uncomment the following lines to take input from the user
#num1 = int(input("Enter first number: "))
#num2 = int(input("Enter second number: "))

print("The L.C.M. of", num1,"and", num2,"is", lcm(num1, num2))

```

Output:

The L.C.M. of 54 and 24 is 216

Question :- Subtract the mean of a row from each element of the row in a Data Frame.

Sol:- To solve this, follow these steps

1) Create a list e.g. d with three columns named a,b,c with some values.

```

d= {
    'a' = [10,12,13],

```

'b' = [20,3,4],

'c' = [30,4,6]

}

2) Then create a dataframe in Panda (which is same as spreadsheet in excel)

using `df= pd.DataFrame(d)`

3) To see the dataframe use command :- `print(df)`

4) Now we have to find the mean value of each row , for that we can use the formula

`m = df.mean(axis=1)`

5) To subtract the mean of a row from each element of the row in a Data Frame.

formula is `df.sub(m, axis=0)` or `df.sub(df.mean(axis=1), axis=0)`

```
In [13]: d= {
          'a': [10,12,13],
          'b': [20,3,4],
          'c': [30,4,6]
        }
df=pd.DataFrame(d)
print(df)
```

```
   a  b  c
0  10 20 30
1  12  3  4
2  13  4  6
```

```
In [14]: df.mean(axis=1)
```

```
Out[14]: 0    20.000000
         1     6.333333
         2     7.666667
         dtype: float64
```

```
In [15]: df.sub(df.mean(axis=1), axis=0)
```

```
Out[15]:
```

```
   a         b         c
0 -10.000000  0.000000  10.000000
1  5.666667 -3.333333 -2.333333
```

DATA FRAME WITH PYTHON

Next to Matplotlib and NumPy, [Pandas](#) is one of the most widely used Python libraries in data science. It is mainly used for data munging, and with good reason: it's very powerful and flexible, among many other things.

The Pandas library has the broader goal of becoming the most powerful and flexible open source data analysis and manipulation tool available in any language.

That's all the more reason for you to get started on working with this library and its expressive data structures straight away!

One of these structures is the DataFrame.

DataFrame. **DataFrame** is a 2-dimensional labeled data structure with columns of potentially different types. You can think of it like a spreadsheet or SQL table, or a dict of Series objects. It is generally the most commonly used **pandas** object.

The concept of a **data frame** comes from the world of statistical software used in empirical research; it generally refers to "tabular" data: a data structure representing cases (rows), each of which consists of a number of observations or measurements (columns)

pandas is a **Python** package providing fast, flexible, and expressive data structures designed to make working with "relational" or "labeled" data both easy and intuitive. It aims to be the fundamental high-level building block for doing practical, real world data analysis in **Python**

R - Data Frames. Advertisements. A **data frame** is a table or a two-dimensional array-like structure in which each column contains values of one variable and each row contains one set of values from each column.

(HOW A DATA FRAME LOOKS)

The diagram shows three input tables on the left, each with a column 'V' and a key column. The 'LEFT' table has keys K0, K1, K2 and values 1, 2, 3. The 'RIGHT' table has keys K0, K0, K3 and values 4, 5, 6. The 'RIGHT2' table has keys K1, K1, K2 and values 7, 8, 9. The 'RESULT' table is a merge of these, with columns V_X, V_Y, and V. It contains rows for keys K0, K0, K1, K1, K2, and K3, with values from the respective input tables. Missing values are represented as NaN.

	V_X	V_Y	V
K0	1.0	4.0	NaN
K0	1.0	5.0	NaN
K1	2.0	NaN	7.0
K1	2.0	NaN	8.0
K2	3.0	NaN	9.0
K3	NaN	6.0	NaN

A data frame is a table or a two-dimensional array-like structure in which each column contains values of one variable and each row contains one set of values from each column.

Following are the characteristics of a data frame.

- The column names should be non-empty.
- The row names should be unique.
- The data stored in a data frame can be of numeric, factor or character type.
- Each column should contain same number of data items.
- Create Data Frame

- **Live Demo**

- **# Create the data frame.**
- `emp.data <- data.frame(`
- `emp_id = c (1:5),`
- `emp_name = c("Rick","Dan","Michelle","Ryan","Gary"),`
- `salary = c(623.3,515.2,611.0,729.0,843.25),`
-
- `start_date = as.Date(c("2012-01-01", "2013-09-23", "2014-11-15", "2014-05-11",`
- `"2015-03-27")),`
- `stringsAsFactors = FALSE`
- `)`
- **# Print the data frame.**
- `print(emp.data)`

- When we execute the above code, it produces the following result –

```
• emp_id emp_name salary start_date
• 1 1 Rick 623.30 2012-01-01
• 2 2 Dan 515.20 2013-09-23
• 3 3 Michelle 611.00 2014-11-15
• 4 4 Ryan 729.00 2014-05-11
• 5 5 Gary 843.25 2015-03-27
```

Expand Data Frame

A data frame can be expanded by adding columns and rows.

Add Column

Just add the column vector using a new column name.

- **# Create the data frame.**

```
emp.data <- data.frame(
  emp_id = c (1:5),
```

```

emp_name = c("Rick","Dan","Michelle","Ryan","Gary"),
salary = c(623.3,515.2,611.0,729.0,843.25),

start_date = as.Date(c("2012-01-01", "2013-09-23", "2014-11-15",
"2014-05-11",
"2015-03-27")),
stringsAsFactors = FALSE
)

# Add the "dept" column.
emp.data$dept <- c("IT","Operations","IT","HR","Finance")
v <- emp.data
print(v)

```

When we execute the above code, it produces the following result –

emp_id	emp_name	salary	start_date	dept
1	Rick	623.30	2012-01-01	IT
2	Dan	515.20	2013-09-23	Operations
3	Michelle	611.00	2014-11-15	IT
4	Ryan	729.00	2014-05-11	HR
5	Gary	843.25	2015-03-27	Finance

Add Row

To add more rows permanently to an existing data frame, we need to bring in the new rows in the same structure as the existing data frame and use the **rbind()** function.

In the example below we create a data frame with new rows and merge it with the existing data frame to create the final data frame.

```
# Create the first data frame.
```

```
emp.data <- data.frame(  
  emp_id = c(1:5),  
  emp_name = c("Rick","Dan","Michelle","Ryan","Gary"),  
  salary = c(623.3,515.2,611.0,729.0,843.25),  
  
  start_date = as.Date(c("2012-01-01", "2013-09-23", "2014-11-15",  
"2014-05-11",  
  "2015-03-27")),  
  dept = c("IT","Operations","IT","HR","Finance"),  
  stringsAsFactors = FALSE  
)
```

```
# Create the second data frame
```

```
emp.newdata <- data.frame(  
  emp_id = c(6:8),  
  emp_name = c("Rasmi","Pranab","Tusar"),  
  salary = c(578.0,722.5,632.8),  
  start_date = as.Date(c("2013-05-21","2013-07-30","2014-06-17")),  
  dept = c("IT","Operations","Fianance"),  
  stringsAsFactors = FALSE  
)
```

```
# Bind the two data frames.
```

```
emp.finaldata <- rbind(emp.data,emp.newdata)
print(emp.finaldata)
```

When we execute the above code, it produces the following result –

```
emp_id  emp_name  salary  start_date  dept
1      1      Rick      623.30    2012-01-01  IT
2      2      Dan       515.20    2013-09-23  Operations
3      3      Michelle  611.00    2014-11-15  IT
4      4      Ryan      729.00    2014-05-11  HR
5      5      Gary      843.25    2015-03-27  Finance
6      6      Rasmi     578.00    2013-05-21  IT
7      7      Pranab    722.50    2013-07-30  Operations
8      8      Tusar     632.80    2014-06-17  Fianance
```

how to find the three largest values in a data frame?

	SUMLEV	REGION	DIVISION	STATE	COUNTY	STNAME	CTYNAME	CENSUS2010POP	ESTIMATESBASE2010	POPESTIMATE2010	...	RDOME
0	40	3	6	1	0	Alabama	Alabama	4779736	4780127	4785161	...	0.002296
1	50	3	6	1	1	Alabama	Autauga County	54571	54571	54660	...	7.242091
2	50	3	6	1	3	Alabama	Baldwin County	182265	182265	183193	...	14.83296
3	50	3	6	1	5	Alabama	Barbour County	27457	27457	27341	...	-4.72813
4	50	3	6	1	7	Alabama	Bibb County	22915	22919	22861	...	-5.52704

For the data set show in the above image, I am trying to find the three most populous states while only taking into consideration the three most populous counties for each state.

I use `CENSUS2010POP`.

This function should return a list of string values(in order of highest population to lowest population).

Below is My Code:

```
x=census_df.groupby('STNAME')['CENSUS2010POP'].nlargest(3)
```

If all your columns are numeric, you can use boolean indexing:

```
In [1]: import pandas as pd
```

```
In [2]: df = pd.DataFrame({'a': [0, -1, 2], 'b': [-3, 2, 1]})
```

```
In [3]: df
```

```
Out[3]:
```

```
   a  b
0  0 -3
1 -1  2
2  2  1
```

```
In [4]: df[df < 0] = 0
```

```
In [5]: df
```

```
Out[5]:
```

```
   a  b
0  0  0
1  0  2
2  2  1
```

```
In [1]: import pandas as pd
```

```
In [2]: df = pd.DataFrame({'a': [0, -1, 2], 'b': [-3, 2, 1],
                          'c': ['foo', 'goo', 'bar']})
```

```
In [3]: df
```

```
Out[3]:
```

```
   a  b  c
0  0 -3 foo
1 -1  2 goo
2  2  1 bar
```

```
0 0 -3 foo
1 -1 2 goo
2 2 1 bar
```

```
In [4]: num = df._get_numeric_data()
```

```
In [5]: num[num < 0] = 0
```

```
In [6]: df
```

```
Out[6]:
   a  b  c
0  0  0  0  foo
1  0  2  2  goo
2  2  1  1  bar
```

With `timedelta` type, boolean indexing seems to work on separate columns, but not on the whole dataframe. So you can do:

```
In [1]: import pandas as pd
```

```
In [2]: df = pd.DataFrame({'a': pd.to_timedelta([0, -1, 2], 'd'),
...:                      'b': pd.to_timedelta([-3, 2, 1], 'd')})
```

```
In [3]: df
```

```
Out[3]:
   a      b
0  0 days -3 days
1 -1 days  2 days
2  2 days  1 days
```

```
In [4]: for k, v in df.iteritems():
```

```
...:     v[v < 0] = 0
...:
```

```
In [5]: df
```

```
Out[5]:
   a      b
0  0 days  0 days
1  0 days  2 days
```

2 2 days 1 days

```
import pandas as pd
```

```
In [20]: df = pd.DataFrame({'a': [-1, 100, -2]})
```

```
In [21]: df
```

```
Out[21]:
```

```
   a
0 -1
1 100
2 -2
```

```
In [22]: df.clip(lower=0)
```

```
Out[22]:
```

```
   a
0  0
1 100
2  0
```

PROGRAM1 on DATA FRAME

```
import pandas as pd
```

```
data = [['Rajiv',10],['Sameer',12],['Kapil',13]]
```

```
df = pd.DataFrame(data,columns=['Name','Age'])
```

```
print (df)
```

```
data1 = {'Name':['Rajiv', 'Sameer', 'Kapil',
```

```
'Nischay'],'Age':[28,34,29,42],
```

```
'Designation':['Accountant','Cashier','Clerk','Manager']}
```

```
df1 = pd.DataFrame(data1)
```

```
print (df1)
```

PROGRAM2 on DATA FRAME

```
import pandas as pd
weather_data={

'day':['01/01/2018','01/02/2018','01/03/2018','01/04/2018','01/0
5/2018','01/01/2018'],
  'temperature':[42,41,43,42,41,40],
  'windspeed':[6,7,2,4,7,2],
  'event':['Sunny','Rain','Sunny','Sunny','Rain','Sunny']
}
df=pd.DataFrame(weather_data)
print(df)

print("Number of Rows and Columns")
print(df.shape)

print(df.head())

print("Tail")
print(df.tail(2))

print("Specified Number of Rows")
print(df[2:5])

print("Print Everything")
print(df[:])

print("Print Column Names")
print(df.columns)

print("Data from Individual Column")
print(df['day']) #or df.day

print(df['temperature'])
print("Maximum Temperature : ", df['temperature'].max())
```



```
print("Printing According to Condition")
print(df[df.temperature>41])

print("Printing the row with maximum temperature")
print(df[df.temperature==df.temperature.max()])

print("Printing specific columns with maximum temperature")
print(df[['day','temperature']][df.temperature==df.temperature.max()])

print("According to index")
print(df.loc[3])

print("Changing of Index")
df.set_index('day',inplace=True)
print(df)

print("Searching according to new index")
print(df.loc['01/03/2018'])

print("Resetting the Index")
df.reset_index(inplace=True)
print(df)

print("Sorting")
print(df.sort_values(by=['temperature'],ascending=False))

print("Sorting on Multiple Columns")
print(df.sort_values(by=['temperature','windspeed'],ascending=True))

print("Sorting on Multiple Columns one in ascending, another in descending")
print(df.sort_values(by=['temperature','windspeed'],ascending=[True,False]))

print("Sum Operations on Data Frame")
print(df['temperature'].sum())
```

```
print("Group By Operations")
print(df.groupby('windspeed')['temperature'].sum())
```

PROGRAM3 on DATA FRAME

```
import pandas as pd
df=pd.read_csv("student.csv", nrows=3)
print("To display selected number of rows from beginning")
print(df)
```

```
df=pd.read_csv("student.csv")
print(df)
```

```
print("Number of Rows and Columns")
print(df.shape)
```

```
print(df.head())
```

```
print("Tail")
print(df.tail(2))
```

```
print("Specified Number of Rows")
print(df[2:5])
```

```
print("Print Everything")
print(df[:])
```

```
print("Print Column Names")
print(df.columns)
```

```
print("Data from Individual Column")
```

```
print(df['Name']) #or df.Name

print(df['Marks'])
print("Maximum Marks : ", df['Marks'].max())

print("Printing According to Condition")
print(df[df.Marks>70])

print("Printing the row with maximum temperature")
print(df[df.Marks==df.Marks.max()])

print("Printing specific columns with maximum Marks")
print(df[['Name', 'Marks']][df.Marks==df.Marks.max()])

print("According to index")
print(df.loc[3])

print("Changing of Index")
df.set_index('Scno',inplace=True)
print(df)

print("Searching according to new index")
print(df.loc[4862])

print("Resetting the Index")
df.reset_index(inplace=True)
print(df)

print("Sorting")
print(df.sort_values(by=['Marks'],ascending=False))

print("Sorting on Multiple Columns")
print(df.sort_values(by=['Class', 'Section'],ascending=True))

print("Sorting on Multiple Columns one in ascending, another in descending")
print(df.sort_values(by=['Marks', 'Name'],ascending=[False, True]))

print("Sum Operations on Data Frame")
```

```
print(df['Marks'].sum())

print("Group By Operations")
print(df.groupby('Class')['Marks'].sum())
```

PROGRAM4 on DATA FRAME

```
import pandas as pd
data = [['Rajiv',10],['Sameer',12],['Kapil',13]]
df = pd.DataFrame(data,columns=['Name','Age'])
print (df)

df.to_csv('new.csv')

df.to_csv('new1.csv', index=False)

df.to_csv('new2.csv', columns=['Name'])

df.to_csv('new4.csv', header=False)
```

PROGRAM5 on DATA FRAME

```
data = np.array(['', 'Col1', 'Col2'],
                ['Row1', 1, 2],
                ['Row2', 3, 4])

print(pd.DataFrame(data=data[1:,1:],
                   index=data[1:,0],
                   columns=data[0,1:]))
```

QUESTIONS & ANSWERS SESSION ON DATA FRAME

What Are Data Frames?

Data frames in Python are very similar: they come with the Pandas library, and they are defined as a two-dimensional labeled data structures with columns of potentially different types.

Data frame as a way to store data in rectangular grids that can easily be overviewed. Each row of these grids corresponds to measurements or values of an instance, while each column is a vector containing data for a specific variable. This means that a data frame's rows do not need to contain, but can contain, the same type of values: they can be numeric, character, logical, etc.

the Pandas data frame consists of three main components: the data, the index, and the columns.

2. How To Select an Index or Column From a Pandas DataFrame

Before you start with adding, deleting and renaming the components of your DataFrame, you first need to know how you can select these elements.

So, how do you do this?

Well, in essence, selecting an index, column or value from your DataFrame isn't that hard. It's really very similar to what you see in other languages that are used for data analysis (and which you might already know!).

Let's take R for example. You use the [,] notation to access the data frame's values. In Pandas DataFrames, this is not too much

different: the most important constructions to use are, without a doubt, `loc` and `iloc`. The subtle differences between these two will be discussed in the next sections.

3. How To Add an Index, Row or Column to a Pandas DataFrame

Now that you have learned how to select a value from a DataFrame, it's time to get to the real work and add an index, row or column to it!

Adding an Index to a DataFrame

When you create a DataFrame, you have the option to add input to the 'index' argument to make sure that you have the index that you desire. When you don't specify this, your DataFrame will have, by default, a numerically valued index that starts with 0 and continues until the last row of your DataFrame.

However, even when your index is specified for you automatically, you still have the power to re-use one of your columns and make it your index. You can easily do this by calling `set_index()` on your DataFrame.

Adding Rows to a DataFrame

Before you can get to the solution, it's first a good idea to grasp the concept of `loc` and how it differs from other indexing attributes such as `iloc` and `ix`:

- `loc` works on labels of your index. This means that if you give in `loc[2]`, you look for the values of your DataFrame that have an index labeled 2.
- `iloc` works on the positions in your index. This means that if you give in `iloc[2]`, you look for the values of your DataFrame that are at index '2'.
- `ix` is a more complex case: when the index is integer-based, you pass a label to `ix`. `ix[2]` then means that you're looking in your DataFrame for values that have an index labeled 2. This is just

like loc! However, if your index is not solely integer-based, ix will work with positions, just like iloc.

Now that the difference between iloc, loc and ix is clear, you are ready to give adding rows to your DataFrame a go!

Adding a Column to Your DataFrame

In some cases, you want to make your index part of your DataFrame. You can easily do this by taking a column from your DataFrame or by referring to a column that you haven't made yet and assigning it to the .index property.

However, if you want to append columns to your DataFrame, you could also follow the same approach as adding an index to your DataFrame: you use loc or iloc.

Note that the observation that was made earlier about loc still stays valid also for when you're adding columns to your DataFrame!

5. How to Rename the Index or Columns of a Pandas DataFrame

To give the columns or your index values of your dataframe a different value, it's best to use the .rename() method.

Tip: try changing the inplace argument in the first task (renaming your columns) to False and see what the script now renders as a result. You see that now the DataFrame hasn't been reassigned when renaming the columns. As a result, the second task takes the original DataFrame as input and not the one that you just got back from the first rename() operation.

6. How To Format The Data in Your Pandas DataFrame

Most of the times, you will also want to be able to do some operations on the actual values that are in your DataFrame.

Replacing All Occurrences of a String in a DataFrame

To replace certain Strings in your DataFrame, you can easily use `replace()`: pass the values that you would like to change, followed by the values you want to replace them by.

8. Does Pandas Recognize Dates When Importing Data?

Pandas can recognize it, but you need to help it a tiny bit: add the argument `parse_dates` when you're reading in data from, let's say, a comma-separated value (CSV) file.

There are, however, always weird date-time formats.

In such cases, you can construct your own parser to deal with this. You could, for example, make a lambda function that takes your `DateTime` and controls it with a format string.

09. How To Write a Pandas DataFrame to a File

When you have done your data munging and manipulation with Pandas, you might want to export the DataFrame to another format. This section will cover two ways of outputting your DataFrame: to a CSV or to an Excel file.

Outputting a DataFrame to CSV

To output a Pandas DataFrame as a CSV file, you can use `to_csv()`.

Writing a DataFrame to Excel

Very similar to what you did to output your DataFrame to CSV, you can use `to_excel()` to write your table to Excel.
