Beginning Python

Ankur Shrivastava

Linux User's Group Manipal

January 29, 2010

Who are we?

- Linux User's Group Manipal
- Life, Universe and FOSS!!
- Believers of Knowledge Sharing
- Most technologically focused "group" in University
- LUG Manipal is a non profit "Group" alive only on voluntary work!!
- `http://lugmanipal.org`

Points To Remember!!!

- If you have problem(s) don't hesitate to ask

Points To Remember!!!

- If you have problem(s) don't hesitate to ask
- Slides are based on Documentation so discussions are really important, slides are for later reference!!

Points To Remember!!!

- If you have problem(s) don't hesitate to ask
- Slides are based on Documentation so discussions are really important, slides are for later reference!!
- Please dont consider sessions as Class (i hate classes)

Points To Remember!!!

- If you have problem(s) don't hesitate to ask
- Slides are based on Documentation so discussions are really important, slides are for later reference!!
- Please dont consider sessions as Class (i hate classes)
- Speaker is just like any person sitting next to you

Points To Remember!!!

- If you have problem(s) don't hesitate to ask
- Slides are based on Documentation so discussions are really important, slides are for later reference!!
- Please dont consider sessions as Class (i hate classes)
- Speaker is just like any person sitting next to you
- Documentation is really important

Points To Remember!!!

- If you have problem(s) don't hesitate to ask
- Slides are based on Documentation so discussions are really important, slides are for later reference!!
- Please dont consider sessions as Class (i hate classes)
- Speaker is just like any person sitting next to you
- Documentation is really important
- Google is your friend

Points To Remember!!!

- If you have problem(s) don't hesitate to ask
- Slides are based on Documentation so discussions are really important, slides are for later reference!!
- Please dont consider sessions as Class (i hate classes)
- Speaker is just like any person sitting next to you
- Documentation is really important
- Google is your friend
- If you have questions after this workshop mail me or come to LUG Manipal's forums
- `http://forums.lugmanipal.org`

What is Python?

- Python is a general purpose, object oriented, high level, interpreted language
- Developed in early 90's by Guido Van Rossum
- Its Simple, Portable, Open Source and Powerfull.

What we will learn?

- History, Features and basic detail
- Language Basics
- Control Flow
- Functions
- Modules
- File I/O

What we require?

- Python interpreter
  Gnu/Linux $->$ Already installed in all distro's
  Mac $->$ Already Installed, if not download `http://python.org`
  Windows $->$ Install Python from Python folder provided to you or download from `http://python.org`

What we require?

- Python interpreter
  Gnu/Linux $->$ Already installed in all distro's
  Mac $->$ Already Installed, if not download `http://python.org`
  Windows $->$ Install Python from Python folder provided to you or download from `http://python.org`
- Text Editor
  Gnu/Linux $->$ Vim/Emacs/Gedit/Kate/Geany any editor will do
  Mac $->$ Vim/TextMate or any other Unix like text editor
  Windows $->$ Notpad++, provided in Notepadpp folder or download `http://notepad-plus.sourceforge.net`
  Don't use Notepad (MS) for editing code, always use Notepad++

What we require?

- Python interpreter
  Gnu/Linux $->$ Already installed in all distro's
  Mac $->$ Already Installed, if not download `http://python.org`
  Windows $->$ Install Python from Python folder provided to you or download from `http://python.org`
- Text Editor
  Gnu/Linux $->$ Vim/Emacs/Gedit/Kate/Geany any editor will do
  Mac $->$ Vim/TextMate or any other Unix like text editor
  Windows $->$ Notpad++, provided in Notepadpp folder or download `http://notepad-plus.sourceforge.net`
  Don't use Notepad (MS) for editing code, always use Notepad++
- Documentation for Python present in Docs folder, web `http://python.org/doc/`
- Set editor to expand tab to 4 spaces.

Where is it used?

- Used extensively in web => Django, TurboGears, Plone, etc
- communicating with Databases => MySQL, PostgreSQL, Oracle, etc
- Desktop GUI => GTK+, QT, Tk, etc
- Scientific Computing => Scipy, Scientific Python, etc
- Network Programming with frameworks/libraries like Twisted, etc
- Software Development => SCons, Buildbot, Roundup, etc
- Games and 3D graphics => pyGame, PyKyra, etc

Difference from C/C++/Java

- No pointers (similar to Java)

Difference from C/C++/Java

- No pointers (similar to Java)
- No prior compilation to Bytecode(?), directly interpreted

Difference from C/C++/Java

- No pointers (similar to Java)
- No prior compilation to Bytecode(?), directly interpreted
- Includes garbage collector(?)

Difference from C/C++/Java

- No pointers (similar to Java)
- No prior compilation to Bytecode(?), directly interpreted
- Includes garbage collector(?)
- Can be used in Procedure(?)/Object Oriented(?) approach/style

Difference from C/C++/Java

- No pointers (similar to Java)
- No prior compilation to Bytecode(?), directly interpreted
- Includes garbage collector(?)
- Can be used in Procedure(?)/Object Oriented(?) approach/style
- English like syntax

Difference from C/C++/Java

- No pointers (similar to Java)
- No prior compilation to Bytecode(?), directly interpreted
- Includes garbage collector(?)
- Can be used in Procedure(?)/Object Oriented(?) approach/style
- English like syntax
- Very good for scripting

Versions of Python

- What do you mean by versions, Python is a language ?

Versions of Python

- What do you mean by versions, Python is a language ?
  Ans) Python as a language keeps on evolving and new features are being added to the language, here by versions we refer to the python interpreter version, new features are added to python interpreter in every release.

Versions of Python

- What do you mean by versions, Python is a language ?
  Ans) Python as a language keeps on evolving and new features are being added to the language, here by versions we refer to the python interpreter version, new features are added to python interpreter in every release.
- important versions are Python 2.6/2.7 and 3.0/3.1
- we will focus on Python 2.6/2.7 and not Python 3.0/3.1
- Python 3.0/3.1 is the future of Python and has non compatible changes from Python 2.X, currently there is less support of Python 3.X and it will take a few years before it matches with that of Python 2.X

Interactive session

- What is interactive session ?

Interactive session

- What is interactive session ?
  Allows you to test your idea/logic/code and play arround with it, works as a shell, you can try out almost anything but whatever you type is note saved.

Interactive session

- What is interactive session ?
  Allows you to test your idea/logic/code and play arround with it, works as a shell, you can try out almost anything but whatever you type is note saved.
- How to start an interactive session ?

Interactive session

- What is interactive session ?
  Allows you to test your idea/logic/code and play arround with it, works as a shell, you can try out almost anything but whatever you type is note saved.
- How to start an interactive session ?
  On Linux open terminal and type `python`
  On Windows, open Python in program files.

Interactive session

- What is interactive session ?
  Allows you to test your idea/logic/code and play arround with it, works as a shell, you can try out almost anything but whatever you type is note saved.
- How to start an interactive session ?
  On Linux open terminal and type `python`
  On Windows, open Python in program files.
- to exit an interactive session type `quit()` or
  press Ctrl + D on Unix like machine
  press Ctrl + Z on Windows machine

Language Basics

Indentation

- In Python indentation is very important.
- There are no end/begin delimiteres like { }
- Grouping of statements are done on basis of their indentation. Statements at same indentation are grouped together in a single block.
- Its recommended to use 4 spaces instead of tabs.
- # marks start of comment (single line)

Indentation

- In Python indentation is very important.
- There are no end/begin delimiteres like { }
- Grouping of statements are done on basis of their indentation. Statements at same indentation are grouped together in a single block.
- Its recommended to use 4 spaces instead of tabs.
- # marks start of comment (single line)

sample code

```
a = 10
if a/10 == 1:
    print ``i think'' # notice 4 spaces before this print
    print ``the value was'' # and this print
    print ``10'' # and this print
```

Numbers

- Integer numbers =>
  decimal -> 1, 44, -44, 2309
  octal -> 01, 022, 077
  hexadecimal -> 0x1, 0x23, 0x3f
  long -> 121212L, 23232329382382938293829382932832938 25L

Numbers

- Integer numbers =>
  decimal -> 1, 44, -44, 2309
  octal -> 01, 022, 077
  hexadecimal -> 0x1, 0x23, 0x3f
  long -> 121212L, 2323232938238293829382938293283293825L
- Floating point => 0.0, 0.32, 2.23e2

Numbers

- Integer numbers =>
  decimal -> 1, 44, -44, 2309
  octal -> 01, 022, 077
  hexadecimal -> 0x1, 0x23, 0x3f
  long -> 121212L, 23232329382382938293829382932832938 25L
- Floating point => 0.0, 0.32, 2.23e2
- Complex numbers => 10+10j, 1+2j, 3-4j where $j = -1^{1/2}$

String

- There is no character data type in python
- Strings can be quoted in single (', “) or triple (''', “““) quotes
- Special characters can be inserted by using the escape character \
- Some commonly used escape sequesces =>
  \\ for a \ in string
  \' for ' in a string
  \" for “ in a string
  \n for a newline
  \r for cariage return
  \t for tab

String

- There is no character data type in python
- Strings can be quoted in single (',“) or triple ("',“““) quotes
- Special characters can be inserted by using the escape character \
- Some commonly used escape sequesces =>
  \\ for a \ in string
  \' for ' in a string
  \" for “ in a string
  \n for a newline
  \r for cariage return
  \t for tab

Example

```
>>>s = 'Line contaning \' and \\ in itself'
>>>print s
Line contaning ' and \ in itself
```

Tuple

- Tuple is an immutable(?) ordered sequence of items(?)

Tuple

- Tuple is an immutable(?) ordered sequence of items(?)
- Tuples can be considered as constant array
- There can have nesting of tuples one inside other
- Elements in a Tuple does not have to be of same type
- Assignment -> `t = (1,2,3,4,‘‘abc’’,2.34,(10,11))`
- Elements can be accessed in way similar to an array

Tuple

- Tuple is an immutable(?) ordered sequence of items(?)
- Tuples can be considered as constant array
- There can have nesting of tuples one inside other
- Elements in a Tuple does not have to be of same type
- Assignment -> `t = (1,2,3,4,‘‘abc’’,2.34,(10,11))`
- Elements can be accessed in way similar to an array

Example

```
>>> t = (1,2,3,4,‘‘abc’’,2.34,(10,11))
>>> t[0]
```

Tuple

- Tuple is an immutable(?) ordered sequence of items(?)
- Tuples can be considered as constant array
- There can have nesting of tuples one inside other
- Elements in a Tuple does not have to be of same type
- Assignment -> `t = (1,2,3,4,‘‘abc’’,2.34,(10,11))`
- Elements can be accessed in way similar to an array

Example

```
>>> t = (1,2,3,4,‘‘abc’’,2.34,(10,11))
>>> t[0]
1
>>> t[4]
```

Tuple

- Tuple is an immutable(?) ordered sequence of items(?)
- Tuples can be considered as constant array
- There can have nesting of tuples one inside other
- Elements in a Tuple does not have to be of same type
- Assignment -> `t = (1,2,3,4,‘‘abc’’,2.34,(10,11))`
- Elements can be accessed in way similar to an array

Example

```
>>> t = (1,2,3,4,‘‘abc’’,2.34,(10,11))
>>> t[0]
1
>>> t[4]
'abc'
>>> t[6]
```

Tuple

- Tuple is an immutable(?) ordered sequence of items(?)
- Tuples can be considered as constant array
- There can have nesting of tuples one inside other
- Elements in a Tuple does not have to be of same type
- Assignment -> `t = (1,2,3,4,‘‘abc’’,2.34,(10,11))`
- Elements can be accessed in way similar to an array

Example

```
>>> t = (1,2,3,4,‘‘abc’’,2.34,(10,11))
>>> t[0]
1
>>> t[4]
'abc'
>>> t[6]
(10,11)
```

List

- List is a mutable ordered sequence of items
- Items in a list can be added or removed
- There can have nesting of lists one inside other
- Elements in a List does not have to be of same type
- Assignment -> `l = [1,2,3,4,‘‘abc’’,2.34,[10,11]]`
- Data access and assignment similar to an array

List

- List is a mutable ordered sequence of items
- Items in a list can be added or removed
- There can have nesting of lists one inside other
- Elements in a List does not have to be of same type
- Assignment -> `l = [1,2,3,4,''abc'',2.34,[10,11]]`
- Data access and assignment similar to an array

Example

```
>>> l = [1,2,3,4,''abc'',2.34,[10,11]]
>>> l[0]
```

List

- List is a mutable ordered sequence of items
- Items in a list can be added or removed
- There can have nesting of lists one inside other
- Elements in a List does not have to be of same type
- Assignment -> `l = [1,2,3,4,‘‘abc’’,2.34,[10,11]]`
- Data access and assignment similar to an array

Example

```
>>> l = [1,2,3,4,‘‘abc’’,2.34,[10,11]]
>>> l[0]
1
>>> l[6]
```

List

- List is a mutable ordered sequence of items
- Items in a list can be added or removed
- There can have nesting of lists one inside other
- Elements in a List does not have to be of same type
- Assignment -> `l = [1,2,3,4,‘‘abc’’,2.34,[10,11]]`
- Data access and assignment similar to an array

Example

```
>>> l = [1,2,3,4,‘‘abc’’,2.34,[10,11]]
>>> l[0]
1
>>> l[6]
[10,11]
>>> l[6][0]
```

List

- List is a mutable ordered sequence of items
- Items in a list can be added or removed
- There can have nesting of lists one inside other
- Elements in a List does not have to be of same type
- Assignment -> `l = [1,2,3,4,''abc'',2.34,[10,11]]`
- Data access and assignment similar to an array

Example

```
>>> l = [1,2,3,4,''abc'',2.34,[10,11]]
>>> l[0]
1
>>> l[6]
[10,11]
>>> l[6][0]
10
```

Dictionaries

- Dictionaries are containers which store items in key/value pairs(?).
- Dictionaries are mutable but does not have any defined sequence.
- Key can be any integer or string and Value can be any item.
- As in Dictionaries values can be accessed by using the key.
- Assignment -> d = { 'key':'value', 1:'value', 'abc':[1,2,3,4] }
- Value can be accessed using the key and keys are unique.

Dictionaries

- Dictionaries are containers which store items in key/value pairs(?).
- Dictionaries are mutable but does not have any defined sequence.
- Key can be any integer or string and Value can be any item.
- As in Dictionaries values can be accessed by using the key.
- Assignment -> d = { 'key':'value', 1:'value', 'abc':[1,2,3,4] }
- Value can be accessed using the key and keys are unique.

Example

```
>>> d = { 'key':'value', 1:'value', 'abc':[1,2,3,4] }
>>> d['key']
```

Dictionaries

- Dictionaries are containers which store items in key/value pairs(?).
- Dictionaries are mutable but does not have any defined sequence.
- Key can be any integer or string and Value can be any item.
- As in Dictionaries values can be accessed by using the key.
- Assignment -> d = { 'key':'value', 1:'value', 'abc':[1,2,3,4] }
- Value can be accessed using the key and keys are unique.

Example

```
>>> d = { 'key':'value', 1:'value', 'abc':[1,2,3,4] }
>>> d['key']
'value'
>>> d[1]
```

Dictionaries

- Dictionaries are containers which store items in key/value pairs(?).
- Dictionaries are mutable but does not have any defined sequence.
- Key can be any integer or string and Value can be any item.
- As in Dictionaries values can be accessed by using the key.
- Assignment -> d = { 'key':'value', 1:'value', 'abc':[1,2,3,4] }
- Value can be accessed using the key and keys are unique.

Example

```
>>> d = { 'key':'value', 1:'value', 'abc':[1,2,3,4] }
>>> d['key']
'value'
>>> d[1]
'value'
>>> d['abc']
```

Dictionaries

- Dictionaries are containers which store items in key/value pairs(?).
- Dictionaries are mutable but does not have any defined sequence.
- Key can be any integer or string and Value can be any item.
- As in Dictionaries values can be accessed by using the key.
- Assignment -> d = { 'key':'value', 1:'value', 'abc':[1,2,3,4] }
- Value can be accessed using the key and keys are unique.

Example

```
>>> d = { 'key':'value', 1:'value', 'abc':[1,2,3,4] }
>>> d['key']
'value'
>>> d[1]
'value'
>>> d['abc']
[1, 2, 3, 4]
```

Index and Slices

- List, Tuple, String, etc can be sliced to get part of data from them.
- Index -> similar to array index refers to position of data.
- Slice -> use to reterive data within particular index.

Example

```
>>> s = “LUG MANIPAL”
>>> s[0]
```

Index and Slices

- List, Tuple, String, etc can be sliced to get part of data from them.
- Index -> similar to array index refers to position of data.
- Slice -> use to reterive data within particular index.

Example

```
>>> s = “LUG MANIPAL”
>>> s[0]
'L'
>>> s[2]
```

Index and Slices

- List, Tuple, String, etc can be sliced to get part of data from them.
- Index -> similar to array index refers to position of data.
- Slice -> use to reterive data within particular index.

Example

```
>>> s = “LUG MANIPAL”
>>> s[0]
'L'
>>> s[2]
'G'
>>> s[0:3]
```

Index and Slices

- List, Tuple, String, etc can be sliced to get part of data from them.
- Index -> similar to array index refers to position of data.
- Slice -> use to reterive data within particular index.

Example

```
>>> s = "LUG MANIPAL"
>>> s[0]
'L'
>>> s[2]
'G'
>>> s[0:3]
'LUG'      from 0 till 3, not including 3!!
>>> s[:3]
```

Index and Slices

- List, Tuple, String, etc can be sliced to get part of data from them.
- Index -> similar to array index refers to position of data.
- Slice -> use to reterive data within particular index.

Example

```
>>> s = "LUG MANIPAL"
>>> s[0]
'L'
>>> s[2]
'G'
>>> s[0:3]
'LUG'      from 0 till 3, not including 3!!
>>> s[:3]
'LUG'      from start till 3, not including 3!!
>>> s[4:]
```

Index and Slices

- List, Tuple, String, etc can be sliced to get part of data from them.
- Index -> similar to array index refers to position of data.
- Slice -> use to reterive data within particular index.

Example

```
>>> s = "LUG MANIPAL"
>>> s[0]
'L'
>>> s[2]
'G'
>>> s[0:3]
'LUG'      from 0 till 3, not including 3!!
>>> s[:3]
'LUG'      from start till 3, not including 3!!
>>> s[4:]
'MANIPAL'      from 4 till end
```

Index and Slices contd.

Example

```
>>> s = "LUG MANIPAL"
>>> s[:11:2]
```

Index and Slices contd.

Example

```
>>> s = “LUG MANIPAL”
>>> s[:11:2]
'LGMNPL'      from start till 11, every 2nd element
>>> s[:11:3]
```

Index and Slices contd.

Example

```
>>> s = "LUG MANIPAL"
>>> s[:11:2]
'LGMNPL'      from start till 11, every 2nd element
>>> s[:11:3]
'L NA'      from start till 11, every 3rd element
>>> s[-1]
```

Index and Slices contd.

Example

```
>>> s = "LUG MANIPAL"
>>> s[:11:2]
'LGMNPL'      from start till 11, every 2nd element
>>> s[:11:3]
'L NA'      from start till 11, every 3rd element
>>> s[-1]
'L'      last element
>>> s[-7:]
```

Index and Slices contd.

Example

```
>>> s = “LUG MANIPAL”
>>> s[:11:2]
'LGMNPL'     from start till 11, every 2nd element
>>> s[:11:3]
'L NA'     from start till 11, every 3rd element
>>> s[-1]
'L'     last element
>>> s[-7:]
'MANIPAL'
>>> s[:-8]
```

Index and Slices contd.

Example

```
>>> s = "LUG MANIPAL"
>>> s[:11:2]
'LGMNPL'      from start till 11, every 2nd element
>>> s[:11:3]
'L NA'      from start till 11, every 3rd element
>>> s[-1]
'L'      last element
>>> s[-7:]
'MANIPAL'
>>> s[:-8]
'LUG'
```

Variables

- There no prior type declaration required for variables.
- A variable can reffer to any Data Type (like Tuple, List, Dictionary, Int, String, Complex, or any other object).

Variables

- There no prior type declaration required for variables.
- A variable can reffer to any Data Type (like Tuple, List, Dictionary, Int, String, Complex, or any other object).
- Variables are references(?) to allocated memory(?).

Variables

- There no prior type declaration required for variables.
- A variable can reffer to any Data Type (like Tuple, List, Dictionary, Int, String, Complex, or any other object).
- Variables are references(?) to allocated memory(?).
- References are always shared(?).

Variables

- There no prior type declaration required for variables.
- A variable can reffer to any Data Type (like Tuple, List, Dictionary, Int, String, Complex, or any other object).
- Variables are references(?) to allocated memory(?).
- References are always shared(?).
- use functions list(old_list) and dict(old_dict) to obtain copy.

Variables

- There no prior type declaration required for variables.
- A variable can reffer to any Data Type (like Tuple, List, Dictionary, Int, String, Complex, or any other object).
- Variables are references(?) to allocated memory(?).
- References are always shared(?).
- use functions list(old_list) and dict(old_dict) to obtain copy.

Note **Be carefull with references they can lead to nasty things!!!**

Variables

- There no prior type declaration required for variables.
- A variable can reffer to any Data Type (like Tuple, List, Dictionary, Int, String, Complex, or any other object).
- Variables are references(?) to allocated memory(?).
- References are always shared(?).
- use functions list(old_list) and dict(old_dict) to obtain copy.

Note **Be carefull with references they can lead to nasty things!!!**

NOTE

```
>>> l = [1, 2, 3, 4]
>>> d = {'key':l}
>>> d['key']
```

Variables

- There no prior type declaration required for variables.
- A variable can reffer to any Data Type (like Tuple, List, Dictionary, Int, String, Complex, or any other object).
- Variables are references(?) to allocated memory(?).
- References are always shared(?).
- use functions list(old_list) and dict(old_dict) to obtain copy.

Note **Be carefull with references they can lead to nasty things!!!**

NOTE

```
>>> l = [1, 2, 3, 4]
>>> d = {'key':l}
>>> d['key']
[1, 2, 3, 4]
```

Variables

- There no prior type declaration required for variables.
- A variable can reffer to any Data Type (like Tuple, List, Dictionary, Int, String, Complex, or any other object).
- Variables are references(?) to allocated memory(?).
- References are always shared(?).
- use functions list(old_list) and dict(old_dict) to obtain copy.

Note **Be carefull with references they can lead to nasty things!!!**

NOTE

```
>>> l = [1, 2, 3, 4]
>>> d = {'key':l}
>>> d['key']
[1, 2, 3, 4]
>>> l[0] = 0
>>> d['key']
```

Variables

- There no prior type declaration required for variables.
- A variable can reffer to any Data Type (like Tuple, List, Dictionary, Int, String, Complex, or any other object).
- Variables are references(?) to allocated memory(?).
- References are always shared(?).
- use functions list(old_list) and dict(old_dict) to obtain copy.

Note **Be carefull with references they can lead to nasty things!!!**

NOTE

```
>>> l = [1, 2, 3, 4]
>>> d = {'key':l}
>>> d['key']
[1, 2, 3, 4]
>>> l[0] = 0
>>> d['key']
[0, 2, 3, 4]
```

Helpful Functions

- help(<obj>) provides help/documentaion for the object using pydoc.help.

Helpful Functions

- help(<obj>) provides help/documentaion for the object using pydoc.help.
- dir(<obj>) lists the attributes/methods avaliable for that object. attributes/methods starting with _/__ are internal attributes/methods and should not be used unless you know what you are doing.

Helpful Functions

- help(<obj>) provides help/documentaion for the object using pydoc.help.
- dir(<obj>) lists the attributes/methods avaliable for that object. attributes/methods starting with _/__ are internal attributes/methods and should not be used unless you know what you are doing.
- dir() returns names in current scope

Helpful Functions

- help(<obj>) provides help/documentaion for the object using pydoc.help.
- dir(<obj>) lists the attributes/methods avaliable for that object. attributes/methods starting with _/__ are internal attributes/methods and should not be used unless you know what you are doing.
- dir() returns names in current scope

Example

```
>>> l = [1, 2, 3]
>>> dir(l)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__delslice__',
'__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__',
'__getslice__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__', '__iter__', '__le__',
'__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__', '__setslice__',
'__sizeof__', '__str__', '__subclasshook__', 'append', 'count', 'extend', 'index',
'insert', 'pop', 'remove', 'reverse', 'sort']
```

List Operations

- l.append($< val >$) $- >$ adds $< val >$ at the end of list.

List Operations

- l.append($< val >$) $- >$ adds $< val >$ at the end of list.
- l.extend($< List >$) $- >$ adds all element in $< list >$ to 'l'.

List Operations

- l.append($< val >$) $- >$ adds $< val >$ at the end of list.
- l.extend($< List >$) $- >$ adds all element in $< list >$ to 'l'.
- l.insert($< pos >, < val >$) $- >$ Inserts $< val >$ at position $< pos >$.

List Operations

- l.append($< val >$) $- >$ adds $< val >$ at the end of list.
- l.extend($< List >$) $- >$ adds all element in $< list >$ to 'l'.
- l.insert($< pos >, < val >$) $- >$ Inserts $< val >$ at position $< pos >$.
- l.remove($< val >$) $- >$ removes first element matching $< val >$, raises ValueError if no such value exists.

List Operations

- l.append($< val >$) $- >$ adds $< val >$ at the end of list.
- l.extend($< List >$) $- >$ adds all element in $< list >$ to 'l'.
- l.insert($< pos >, < val >$) $- >$ Inserts $< val >$ at position $< pos >$.
- l.remove($< val >$) $- >$ removes first element matching $< val >$, raises ValueError if no such value exists.
- l.index($< val >$) $- >$ returns index of first occurence of $< val >$, raises ValueError if no such value exists.

List Operations

- l.append($< val >$) $- >$ adds $< val >$ at the end of list.
- l.extend($< List >$) $- >$ adds all element in $< list >$ to 'l'.
- l.insert($< pos >, < val >$) $- >$ Inserts $< val >$ at position $< pos >$.
- l.remove($< val >$) $- >$ removes first element matching $< val >$, raises ValueError if no such value exists.
- l.index($< val >$) $- >$ returns index of first occurence of $< val >$, raises ValueError if no such value exists.
- l.pop($< index >$) $- >$ removes element at $< index >$, if no index is specified last element is returned.

List Operations

- l.append($< val >$) $- >$ adds $< val >$ at the end of list.
- l.extend($< List >$) $- >$ adds all element in $< list >$ to 'l'.
- l.insert($< pos >, < val >$) $- >$ Inserts $< val >$ at position $< pos >$.
- l.remove($< val >$) $- >$ removes first element matching $< val >$, raises ValueError if no such value exists.
- l.index($< val >$) $- >$ returns index of first occurence of $< val >$, raises ValueError if no such value exists.
- l.pop($< index >$) $- >$ removes element at $< index >$, if no index is specified last element is returned.

Example

```
>>> l = [1, 2, 3, 4, 5, 6, 7]
>>> l.append(8)
>>> l
[1, 2, 3, 4, 5, 6, 7, 8]
```

Dictionary Operations

- d.has_key(< *val* >) − > returns true if key by < *val* > exists, else returns false.
- d.items() − > returns list of 2 value tuple, with first element key and second value.
- d.keys() − > returns list of all keys in dictionary.
- d.values() − > returns list of all values in dictionary.
- d.iteritems() − > returns an iteratable object of dictionay, giving a tuple of (key, value) on every iteration.

Example

```
>>> d = { 'a':1 , 'abc':878 }
>>> for i,j in d.iteritems():
...     print i, j
...
a 1
abc 878
```

Control Flow

Input and Output

Input

- to take input (string) from user we use function raw_input().
- function input() evaluates the input as python expression.
- we use functions int(), long(), float(), and str() to convert the input to desired type.

Input and Output

Input

- to take input (string) from user we use function raw_input().
- function input() evaluates the input as python expression.
- we use functions int(), long(), float(), and str() to convert the input to desired type.

Output

- print is a keyword for giving output to a console or a file.
- print can take multiple arguments saperated by comma (,)
- if you dont want a newline add comma (,) at the end.

Input and Output

Input

- to take input (string) from user we use function raw_input().
- function input() evaluates the input as python expression.
- we use functions int(), long(), float(), and str() to convert the input to desired type.

Output

- print is a keyword for giving output to a console or a file.
- print can take multiple arguments saperated by comma (,)
- if you dont want a newline add comma (,) at the end.

Example

```
>>> val = raw_input("Enter a number: ")
Enter a number: 123
>>> val = int(val) + 1
>>> print "Number is", val
Number is 124
```

if

- if is a conditional keyword, for a simple “if then else” clause in english.
- Header lines(?) are always concluded with a “:” followed by indented block of statements.
- optionally if can be followed by an “else if” which is known as “elif” in Python.
- expressions can be logically connected by using “or”/“and”.

Tip Just remember we need to put “:” where every we used “{}” in other languages, and statements following 'if' should always be indented.

if

- if is a conditional keyword, for a simple “if then else” clause in english.
- Header lines(?) are always concluded with a “:” followed by indented block of statements.
- optionally if can be followed by an “else if” which is known as “elif” in Python.
- expressions can be logically connected by using “or”/“and”.

Tip Just remember we need to put “:” where every we used “{}” in other languages, and statements following 'if' should always be indented.

Example

```
if a == 1:
    print “value of a is 1”
elif a == 2:
    print “value of a is 2”
else:
    print “value of a is not 1 or 2”
```

while

- while is used for repeated execution of a block of code till a condition holds true.
- in Python while has an optional else clause which executes when the condition evaluates to false.
- following values are considered flase -> None, False, any numeric type equal to zero, any empty sequence (), [],'' or any empty mapping {}.

Example

```
limit = 5
val = 0
while val < limit :
    print val,
    val +=1
```

Output :

while

- while is used for repeated execution of a block of code till a condition holds true.
- in Python while has an optional else clause which executes when the condition evaluates to false.
- following values are considered flase -> None, False, any numeric type equal to zero, any empty sequence (), [],'' or any empty mapping {}.

Example

```
limit = 5
val = 0
while val < limit :
    print val,
    val +=1
```

Output :

```
0 1 2 3 4
```

for

- for is a sequence iterator(?).

for

- for is a sequence iterator(?).
- for works on strings, lists, tuples, etc.
- use range/xrange to generate lists
- range(0,4) $->$ [0, 1, 2, 3]
  range(0,6,2) $->$ [0, 2, 4]
- xrange works as a iterator and does not generate a list (?)

for

- for is a sequence iterator(?).
- for works on strings, lists, tuples, etc.
- use range/xrange to generate lists
- range(0,4) $->$ [0, 1, 2, 3]
  range(0,6,2) $->$ [0, 2, 4]
- xrange works as a iterator and does not generate a list (?)

Example

```
for item in range(1, 5) :
    print item,
```

Output:

for

- for is a sequence iterator(?).
- for works on strings, lists, tuples, etc.
- use range/xrange to generate lists
- range(0,4) $->$ [0, 1, 2, 3]
  range(0,6,2) $->$ [0, 2, 4]
- xrange works as a iterator and does not generate a list (?)

Example

```
for item in range(1, 5) :
    print item,
```

Output:

```
1 2 3 4
```

break / continue

break

- used for loop termination
- if nested, terminates the inner loop

continue

- terminates the current iteration and starts the next
- it does not terminate the loop

break / continue

break

- used for loop termination
- if nested, terminates the inner loop

continue

- terminates the current iteration and starts the next
- it does not terminate the loop

Example

```
for i in range(1,10) :
    if i == 6 :
        break
    if i == 3 :
        continue
    print i,
```

Output:

break / continue

break

- used for loop termination
- if nested, terminates the inner loop

continue

- terminates the current iteration and starts the next
- it does not terminate the loop

Example

```
for i in range(1,10) :
    if i == 6 :
        break
    if i == 3 :
        continue
    print i,
```

Output:

```
1 2 4 5
```

List Comprehensions

- Normal use of “for” loop is to iterate and build a new list
- List comprehensions simplifies the above task
- Syntax − >
  [<expression> for <target> in <iterable> <condiction>]
- there can be multiples statements.

Example

```
>>> [ x*2 for x in range(1,5) ]
```

List Comprehensions

- Normal use of "for" loop is to iterate and build a new list
- List comprehensions simplifies the above task
- Syntax − >
  [<expression> for <target> in <iterable> <condiction>]
- there can be multiples statements.

Example

```
>>> [ x*2 for x in range(1,5) ]
[2, 4, 6, 8]
>>> [ x for x in range(0,10) if x%2 == 0 and x > 2]
```

List Comprehensions

- Normal use of “for” loop is to iterate and build a new list
- List comprehensions simplifies the above task
- Syntax − >
  [<expression> for <target> in <iterable> <condiction>]
- there can be multiples statements.

Example

```
>>> [ x*2 for x in range(1,5) ]
[2, 4, 6, 8]
>>> [ x for x in range(0,10) if x%2 == 0 and x > 2]
[4, 6, 8]
>>> [ x+y for x in range(1,4) for y in range(1,4)]
```

List Comprehensions

- Normal use of “for” loop is to iterate and build a new list
- List comprehensions simplifies the above task
- Syntax − >
  [<expression> for <target> in <iterable> <condition>]
- there can be multiples statements.

Example

```
>>> [ x*2 for x in range(1,5) ]
[2, 4, 6, 8]
>>> [ x for x in range(0,10) if x%2 == 0 and x > 2]
[4, 6, 8]
>>> [ x+y for x in range(1,4) for y in range(1,4)]
[2, 3, 4, 3, 4, 5, 4, 5, 6]
```

Functions

What are Functions?

- A function is a group of statements that executes on request.
- In Python functions are also objects.
- function return type is not required.
- if function does not return any value, default value of None is returned.
- a function can take another function name as argument and return a function name (as in functional programming languages).
- a function is defined using the keyoword def followed by function name and parameters

What are Functions?

- A function is a group of statements that executes on request.
- In Python functions are also objects.
- function return type is not required.
- if function does not return any value, default value of None is returned.
- a function can take another function name as argument and return a function name (as in functional programming languages).
- a function is defined using the keyoword def followed by function name and parameters

Example

```
>>> def abc(arg):
...       print arg
...
>>> abc("Hello")
Hello
```

Parameters

- Default Value is the value assigned to function argument in function defination.
- Types of Parameters

Mandatory	Parameters with no default values.
Optional	Parameters with default values.

- At function call values for all mandatory parameters are required.
- There is no function overloading in python.

Parameters

- Default Value is the value assigned to function argument in function defination.
- Types of Parameters

Mandatory	Parameters with no default values.
Optional	Parameters with default values.

- At function call values for all mandatory parameters are required.
- There is no function overloading in python.

Example

```
def abc (arg1,arg2=10): # arg2 has default value of 10
    print arg1, arg2
abc(1)
abc(2,3)
```

Output :

Parameters

- Default Value is the value assigned to function argument in function defination.
- Types of Parameters

Mandatory	Parameters with no default values.
Optional	Parameters with default values.

- At function call values for all mandatory parameters are required.
- There is no function overloading in python.

Example

```
def abc (arg1,arg2=10): # arg2 has default value of 10
    print arg1, arg2
abc(1)
abc(2,3)
```

Output :

```
1 10
```

Parameters

- Default Value is the value assigned to function argument in function defination.
- Types of Parameters

Mandatory	Parameters with no default values.
Optional	Parameters with default values.

- At function call values for all mandatory parameters are required.
- There is no function overloading in python.

Example

```
def abc (arg1,arg2=10): # arg2 has default value of 10
    print arg1, arg2
abc(1)
abc(2,3)
```

Output :

```
1 10
2 3
```

Modules

What are Modules?

- Modules group code and data for reuse.
- Modules or a part of modules can be used in other code using *import* or *from* statements.

What are Modules?

- Modules group code and data for reuse.
- Modules or a part of modules can be used in other code using *import* or *from* statements.

import

- Syntax $->$
  import module [*as othername*]
- *import* imports whole of specified module in the namespace module/othername.

What are Modules?

- Modules group code and data for reuse.
- Modules or a part of modules can be used in other code using *import* or *from* statements.

import

- Syntax $->$
  import module [*as othername*]
- *import* imports whole of specified module in the namespace module/othername.

from

- Syntax $->$
  from module import something [*as somethingelse*]
- using *from* import something inside the current namespace as something/somethingelse.

How to make Module?

- Any python file (*.py*) can work as a module.
- If the file is written to execute when invoked, it is executed when imported.
- To allow a file to executed when invoked and avoid when imported we compare variable "__name__"
- Python file executing as main code has variable "__name__" set to "__main__"
- Python file executing as module has variable "__name__" set to the module name

Lets create a file example.py, which we will use to describe modules

example.py

```
# some functions
def div(a,b):
    print a/b
# code that will execute in every case
print "Hi"
# code that will execute only if file invoked
if __name__ == "__main__":
    mul(2,2)
    print "not as module"
else:
    print "as module"
```

Try *import* and *from* on example.py
and also try executing the file.

File I/O

File

- *file* is a built-in type in Python.
- Allows access to files in an Operating System independent manner.
- Different modes to access a file are

File

- *file* is a built-in type in Python.
- Allows access to files in an Operating System independent manner.
- Different modes to access a file are
  - 'r' opens an already existing file in read only mode.

File

- *file* is a built-in type in Python.
- Allows access to files in an Operating System independent manner.
- Different modes to access a file are
    - 'r' opens an already existing file in read only mode.
    - 'w' opens a file in write only mode, if file exists it is truncated else a new file in created.

File

- *file* is a built-in type in Python.
- Allows access to files in an Operating System independent manner.
- Different modes to access a file are
  - 'r' opens an already existing file in read only mode.
  - 'w' opens a file in write only mode, if file exists it is truncated else a new file in created.
  - 'a' opens a file in write only mode, if file exists new data in added at the end, else a new file is created.

File

- *file* is a built-in type in Python.
- Allows access to files in an Operating System independent manner.
- Different modes to access a file are
  - 'r' opens an already existing file in read only mode.
  - 'w' opens a file in write only mode, if file exists it is truncated else a new file in created.
  - 'a' opens a file in write only mode, if file exists new data in added at the end, else a new file is created.
  - 'r+' file is opened in read and write mode, but file must exist.

File

- *file* is a built-in type in Python.
- Allows access to files in an Operating System independent manner.
- Different modes to access a file are
  - 'r' opens an already existing file in read only mode.
  - 'w' opens a file in write only mode, if file exists it is truncated else a new file in created.
  - 'a' opens a file in write only mode, if file exists new data in added at the end, else a new file is created.
  - 'r+' file is opened in read and write mode, but file must exist.
  - 'w+' file is opened in read and write mode, file is truncated if exists, else a new file is created.

File

- *file* is a built-in type in Python.
- Allows access to files in an Operating System independent manner.
- Different modes to access a file are
  - 'r' opens an already existing file in read only mode.
  - 'w' opens a file in write only mode, if file exists it is truncated else a new file in created.
  - 'a' opens a file in write only mode, if file exists new data in added at the end, else a new file is created.
  - 'r+' file is opened in read and write mode, but file must exist.
  - 'w+' file is opened in read and write mode, file is truncated if exists, else a new file is created.
  - 'a+' file is opened in read and write mode, if file exists new data in added at the end, else a new file is created.

File

- *file* is a built-in type in Python.
- Allows access to files in an Operating System independent manner.
- Different modes to access a file are
  - 'r' opens an already existing file in read only mode.
  - 'w' opens a file in write only mode, if file exists it is truncated else a new file in created.
  - 'a' opens a file in write only mode, if file exists new data in added at the end, else a new file is created.
  - 'r+' file is opened in read and write mode, but file must exist.
  - 'w+' file is opened in read and write mode, file is truncated if exists, else a new file is created.
  - 'a+' file is opened in read and write mode, if file exists new data in added at the end, else a new file is created.
  - append 'b' to the mode to open file in binary mode.

File

- *file* is a built-in type in Python.
- Allows access to files in an Operating System independent manner.
- Different modes to access a file are
  - 'r' opens an already existing file in read only mode.
  - 'w' opens a file in write only mode, if file exists it is truncated else a new file in created.
  - 'a' opens a file in write only mode, if file exists new data in added at the end, else a new file is created.
  - 'r+' file is opened in read and write mode, but file must exist.
  - 'w+' file is opened in read and write mode, file is truncated if exists, else a new file is created.
  - 'a+' file is opened in read and write mode, if file exists new data in added at the end, else a new file is created.
  - append 'b' to the mode to open file in binary mode.
- modes are passed to function *open* which is used to open a file
  Synatx − >
  <file obj> = open(<file name>,<mode>,bufsize=-1)

File Object Methods

Let us assume we have
f = open('file','r+')

File Object Methods

Let us assume we have
f = open('file','r+')

- f.read([size]) reads size bytes from file and returns, if size is not specified or if size < 0 then size reads whole file.

File Object Methods

Let us assume we have
f = open('file','r+')

- f.read([size]) reads size bytes from file and returns, if size is not specified or if size $<$ 0 then size reads whole file.
- f.readline() reads one line from file (till \n) and returns the line (including \n)

File Object Methods

Let us assume we have
f = open('file','r+')

- f.read([size]) reads size bytes from file and returns, if size is not specified or if size $<$ 0 then size reads whole file.
- f.readline() reads one line from file (till \n) and returns the line (including \n)
- f.readlines() reads all lines in file and returns a list of all lines.

File Object Methods

Let us assume we have
f = open('file','r+')

- f.read([size]) reads size bytes from file and returns, if size is not specified or if size $<$ 0 then size reads whole file.
- f.readline() reads one line from file (till \n) and returns the line (including \n)
- f.readlines() reads all lines in file and returns a list of all lines.
- f.write(data) writes data to file.

File Object Methods

Let us assume we have
f = open('file','r+')

- f.read([size]) reads size bytes from file and returns, if size is not specified or if size $<$ 0 then size reads whole file.
- f.readline() reads one line from file (till \n) and returns the line (including \n)
- f.readlines() reads all lines in file and returns a list of all lines.
- f.write(data) writes data to file.
- f.close() closes the file.

And we are done..

Thank You!!!

And we are done..

Questions?

Contact Information

Ankur Shrivastava
ankur@ankurs.com
`http://ankurs.com`

Linux User's Group Manipal
`http://lugmanipal.org`
`http://forums.lugmanipal.org`

Copying