

# **Python Workshop**

**Baiju Muthukadan  
ZeOmega, Bangalore**

**FOSSMeet'14, NIT Calicut**

**Feb 15, 2014**

# Prerequisites

- **GNU/Linux**
- **Python 2.7**
- **Text editor**

# **Expectation from Participants**

**Familiarity with programming in another language (C/C++/Java/C#/Ruby/PHP)**

# About Me

- **Founded the SMC project in 2001 while studying at REC Calicut**
- **Employed by FSF India in 2002-2003**
- **Contributor to Zope project**
- **Book Author: A Comprehensive Guide to Zope Component Architecture**
- **During PyCon India 2013, received the first Kenneth Gonsalves Award**

# Attribution

**This presentation and exercises are based on:**

`http://tdc-www.harvard.edu/Python.pdf`

`http://en.wikibooks.org/wiki/Non-Programmer's\_Tutorial\_for\_Python\_2.6`

# Introduction

- **Free/Open source general-purpose language**
- **Multi-paradigm -- Object Oriented, Procedural, Functional**
- **Easy to interface with C/C++/ObjC/Java/Fortran**
- **Great interactive environment**
- **very clear, readable syntax**
- **strong introspection capabilities**
- **intuitive object orientation**
- **full modularity, supporting hierarchical packages**
- **exception-based error handling**
- **very high level dynamic data types**
- **extensive standard libraries and third party modules for virtually every task**
- **embeddable within applications as a scripting interface**

# Python Version

- **"Current" version is 3.3**
- **"Mainstream" version is 2.7**
- **Use 3.3 if dependencies are available, otherwise use 2.7**

# Installation

- **Python comes pre-installed with GNU/Linux and Mac**
- **Windows binaries from <http://python.org/>**



# Python Interactive Interpreter

- **Interactive interface to Python**

```
$ python
```

```
Python 2.7.5 (default, Nov 12 2013, 16:45:54)
```

```
[GCC 4.8.2 20131017 (Red Hat 4.8.2-1)] on linux2
```

```
Type "help", "copyright", "credits" or "license" for more info
```

```
>>>
```

- **Python interpreter evaluates inputs**

```
>>> 3 * (7 + 2)
```

```
27
```

```
>>> 'Hello ' + 'World!'
```

```
'Hello World!'
```

# Python Interactive Interpreter - Continued

- Python prompts with "`>>>`" (Primary) and "`...` " (Secondary)

```
>>> if 1 < 2:  
...     print "1 is less than 2"  
...  
1 is less than 2
```

- To exit Python interactive interpreter : CTRL+D

# Running Programs on GNU/Linux

- **Easy way to run a program:**

```
$ python filename.py
```

- **You could make the \*.py file executable and add "#!/usr/bin/env python" to the top of that file to make it run.**

```
$ ./filename.py
```

# Batteries Included

**Large collection of proven modules are included in the standard distribution:**

`http://docs.python.org/2/library/index.html`

**And many more third part packages are available from PyPI (Python Package Index Server aka. Cheeseshop):**

`https://pypi.python.org/pypi`

# A Code Sample

```
x = 34 - 23 # A comment.  
y = "Hello" # Another one.  
z = 3.45  
if z == 3.45 or y == "Hello":  
    x = x + 1  
    y = y + " World" # String concat.  
print x  
print y
```

# Enough to Understand the Code

- **Assignment uses = and comparison uses ==.**
- **For numbers + - \* / % are as expected.**
  - ▶ **Special use of + for string concatenation.**
  - ▶ **Special use of % for string formatting (as with *printf* in C)**
- **Logical operators are words (and, or, not) not symbols**
- **The basic printing command is *print*.**
- **The first assignment to a variable creates it.**
  - ▶ **Variable types don't need to be declared.**
  - ▶ **Python figures out the variable types on its own.**

# Basic Data types

- **Integers (default for numbers)  $z = 5 / 2$  # Answer is 2, integer division.**
- **Floats  $x = 3.456$**
- **Strings**
- **Can use " " or ' ' to specify. "abc" 'abc' (Same thing.)**
- **Unmatched can occur within the string. "matt's"**
- **Use triple double-quotes for multi-line strings or strings than contain both ' and " inside of them: """a'b"c"""**



# White space

**White space is meaningful in Python: especially indentation and placement of newlines.**

- **Use a newline to end a line of code.**
  - ▶ **Use `\` when must go to next line prematurely.**
- **No braces { } to mark blocks of code in Python... use consistent indentation instead.**
  - ▶ **The first line with less indentation is outside of the block.**
  - ▶ **The first line with more indentation starts a nested block**
- **Often a colon appears at the start of a new block. (E.g. for function definitions and if conditions.)**



# Comments

- **Start comments with # - the rest of line is ignored.**
- **Can include a "documentation string" as the first line of any new function or class that you define.**
- **The development environment, debugger, and other tools use it: it's good style to include one.**

```
def my_function(x, y):  
    """This is the docstring. This  
    function does blah blah blah."""  
    # The code would go here...
```

# Assignment

- **Binding a variable in Python means setting a name to hold a reference to some object.**
  - ▶ **Assignment creates references, not copies**
- **Names in Python do not have an intrinsic type. Objects have types.**
  - ▶ **Python determines the type of the reference automatically based on the data object assigned to it.**
- **You create a name the first time it appears on the left side of an assignment expression:**  
$$x = 3$$
- **A reference is deleted via garbage collection after any names bound to it have passed out of scope.**

# Accessing Non-Existent Names

- **If you try to access a name before it's been properly created (by placing it on the left side of an assignment), you'll get an error.**

```
>>> y
```

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

```
  File "<stdin>", line 1, in <module>
```

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

```
>>> y = 3
```

```
>>> y
```

```
3
```

# Multiple Assignment

- **You can also assign to multiple names at the same time.**

```
>>> x, y = 2, 3
```

```
>>> x
```

```
2
```

```
>>> y
```

```
3
```

# Naming Rules

- **Names are case sensitive and cannot start with a number. They can contain letters, numbers, and underscores.**

bob Bob \_bob \_2\_bob\_ bob\_2 BoB

- **There are some reserved words:**

and, assert, break, class, continue, def, del, elif, else, except, exec, finally, for, from, global, if, import, in, is, lambda, not, or, pass, print, raise, return, try, while

# User Input

- Use *raw\_input* function to get user input as a string.  
`name = raw_input("Enter name: ")`
- Use *input* function to get user input with evaluation of the given expression  
`age = input("Enter age: ")`

# Control of Flow - if conditions

- **Branching logic can be implemented using if, elif & else keywords.**
- **There is no "switch" statement in Python.**

```
if x == 3:  
    print "X equals 3."  
elif x == 2:  
    print "X equals 2."  
else:  
    print "X equals something else."  
  
print "This is outside the 'if'."
```

# Control of Flow - while loop

```
while x < 10:  
    if x > 7:  
        x += 2  
        continue  
    x = x + 1  
    print "Still in the loop."  
    if x == 8:  
        break  
print "Outside of the loop."
```



# Control of Flow - for loop

```
for x in range(10):  
  
    if x > 7:  
        x += 2  
        continue  
    x = x + 1  
    print "Still in the loop."  
    if x == 8:  
        break  
  
print "Outside of the loop."
```

# Sequence Types

## 1. Tuple

- ▶ A simple immutable ordered sequence of items
- ▶ Items can be of mixed types, including collection types

## 2. Strings

- ▶ Immutable
- ▶ Conceptually very much like a tuple

## 3. List

- ▶ Mutable ordered sequence of items of mixed types

# Similar Syntax

- **All three sequence types (tuples, strings, and lists) share much of the same syntax and functionality.**
- **Key difference:**
  - ▶ **Tuples and strings are immutable**
  - ▶ **Lists are mutable**
- **The operations shown in this section can be applied to all sequence types**
  - ▶ **most examples will just show the operation performed on one**

# Sequence Types 1

- **Tuples are defined using parentheses (and commas).**

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

- **Lists are defined using square brackets (and commas).**

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

- **Strings are defined using quotes (" , ' , '' , ''").**

```
>>> st = "Hello World"
```

```
>>> st = 'Hello World'
```

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

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

## Sequence Types 2

- We can access individual members of a tuple, list, or string using square bracket "array" notation.
- Note that all are 0 based...

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

```
>>> tu[1] # Second item in the tuple.  
'abc'
```

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

```
>>> li[1] # Second item in the list.  
34
```

```
>>> st = "Hello World"
```

```
>>> st[1] # Second character in string.  
'e'
```

# Positive and negative indices

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

**Positive index: count from the left, starting with 0.**

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

**Negative lookup: count from right, starting with -1.**

```
>>> t[-3]
4.56
```

# Slicing: Return Copy of a Subset 1

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

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

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

**You can also use negative indices when slicing.**

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



## Slicing: Return Copy of a Subset 2

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

**Omit the first index to make a copy starting from the beginning of the container.**

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

**Omit the second index to make a copy starting at the first index and going to the end of the container.**

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



# Copying the Whole Sequence

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

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

**Note the difference between these two lines for mutable sequences:**

```
>>> list2 = list1 # 2 names refer to 1 ref
```

**# Changing one affects both**

```
>>> list2 = list1[:] # Two independent copies, two refs
```

# The 'in' Operator

- **Boolean test whether a value is inside a container:**

```
>>> t = [1, 2, 4, 5]
```

```
>>> 3 in t
```

```
False
```

```
>>> 4 in t
```

```
True
```

```
>>> 4 not in t
```

```
False
```

- **For strings, tests for sub-strings**

```
>>> a = 'abcde'
```

```
>>> 'c' in a
```

```
True
```

```
>>> 'cd' in a
```

```
True
```

```
>>> 'ac' in a
```

```
False
```

- ***in* keyword is used in the *for* loops and list comprehensions.**

# The + Operator

- **The + operator produces a new tuple, list, or string whose value is the concatenation of its arguments.**

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

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

```
>>> [1, 2, 3] + [4, 5, 6]
```

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

```
>>> "Hello" + " " + "World"
```

```
'Hello World'
```

# The \* Operator

- The \* operator produces a new tuple, list, or string that "repeats" the original content.

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

# Tuples: Immutable

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

```
>>> t[2] = 3.14
```

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

```
  File "<stdin>", line 1, in <module>
```

```
TypeError: 'tuple' object does not support item assignment
```

**You can't change a tuple. You can make a fresh tuple and assign its reference to a previously used name.**

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

# Lists: Mutable

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

- **We can change lists in place.**
- **Name *li* still points to the same memory reference when we're done.**
- **The mutability of lists means that they aren't as fast as tuples.**

# Operations on Lists Only 1

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

# The extend method vs the + operator

- **+ creates a fresh list (with a new memory reference)**
- **extend operates on list li in place.**

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

## **Confusing:**

- **Extend takes a list as an argument.**
- **Append takes a singleton as an argument.**

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



# Operations on Lists Only 3

```
>>> li = ['a', 'b', 'c', 'b']
>>> li.index('b') # index of first occurrence
1
>>> li.count('b') # number of occurrences
2
>>> li.remove('b') # remove first occurrence
>>> li
['a', 'c', 'b']
```

# Operations on Lists Only 4

```
>>> li = [5, 2, 6, 8]
>>> li.reverse() # reverse the list *in place*
>>> li
[8, 6, 2, 5]
>>> li.sort() # sort the list *in place*
>>> li
[2, 5, 6, 8]
>>> li.sort(some_function)
# sort in place using user-defined comparison
```

# Tuples vs. Lists

- **Lists slower but more powerful than tuples.**
- **Lists can be modified, and they have lots of handy operations we can perform on them.**
- **Tuples are immutable and have fewer features.**
- **To convert between tuples and lists use the `list()` and `tuple()` functions:**

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

# Dictionaries: A Mapping type

- **Dictionaries store a mapping between a set of keys and a set of values.**
- **Keys can be any immutable type.**
- **Values can be any type**
- **A single dictionary can store values of different types**
- **You can define, modify, view, lookup, and delete the key-value pairs in the dictionary.**

# Using dictionaries

```
>>> d = {'user':'bozo', 'pswd':1234}
```

```
>>> d['user']
```

```
'bozo'
```

```
>>> d['pswd']
```

```
1234
```

```
>>> d['bozo']
```

```
Traceback (innermost last):
```

```
File '<interactive input>' line 1, in ?
```

```
KeyError: bozo
```

```
>>> d = {'user':'bozo', 'pswd':1234}
```

```
>>> d['user'] = 'clown'
```

```
>>> d
```

```
{'user':'clown', 'pswd':1234}
```

```
>>> d['id'] = 45
```

```
>>> d
```

```
{'user':'clown', 'id':45, 'pswd':1234}
```

## Using dictionaries - Continued

```
>>> d = {'user':'bozo', 'p':1234, 'i':34}
>>> del d['user'] # Remove one.
>>> d
{'p':1234, 'i':34}
>>> d.clear() # Remove all.
>>> d
{}
>>> d = {'user':'bozo', 'p':1234, 'i':34}
>>> d.keys() # List of keys.
['user', 'p', 'i']
>>> d.values() # List of values.
['bozo', 1234, 34]
>>> d.items() # List of item tuples.
[('user','bozo'), ('p',1234), ('i',34)]
```

# Functions

- *def* creates a function and assigns it a name
- *return* sends a result back to the caller
- Arguments are passed by assignment
- Arguments and return types are not declared

```
def <name>(arg1, arg2, ..., argN):  
    <statements>  
    return <value>
```

```
def times(x,y):  
    return x*y
```



# Passing Arguments to Functions

- **Arguments are passed by assignment**
- **Passed arguments are assigned to local names**
- **Assignment to argument names don't affect the caller**
- **Changing a mutable argument will affect the caller and it may not be the expected behavior**

```
def changer(x,y):  
    x = 2 # changes local value of x only  
    y[0] = 'hi' # changes shared object
```



# Optional Arguments

- **Can define defaults for arguments that need not be passed**

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

# Gotchas

- **All functions in Python have a return value**
  - ▶ even if no return line inside the code.
- **Functions without a return return the special value *None*.**
- **There is no function overloading in Python.**
  - ▶ Two different functions can't have the same name, even if they have different arguments.
- **Functions can be used as any other data type. They can be:**
  - ▶ Arguments to function
  - ▶ Return values of functions
  - ▶ Assigned to variables
  - ▶ Parts of tuples, lists, etc

# Why Use Modules?

- **Code reuse**
  - ▶ **Routines can be called multiple times within a program**
  - ▶ **Routines can be used from multiple programs**
- **Namespace partitioning**
  - ▶ **Group data together with functions used for that data**
- **Implementing shared services or data**
  - ▶ **Can provide global data structure that is accessed by multiple subprograms**

# Modules

- **Modules are functions and variables defined in separate files**
- **Items are imported using *from* or *import***

```
from module import function  
function()
```

```
import module  
module.function()
```

- **Modules are namespaces**
- **Can be used to organize variable names, i.e.**

```
atom.position = atom.position - molecule.position
```

# String Formatting

- **Substitute values using a tuple**

```
coins, amount, name = 2, 2.4, 'Tom'  
out = '%s has %d coins worth a total of $%.02f' % (name,  
                                                    coins, amount)  
  
print out
```

```
# Output: 'Tom has 2 coins worth a total of $2.40'
```

- **Substitute value using a dictionary**

```
data = {'coins': 2, 'amount': 2.4, 'name': 'Tom'}  
out = '%(name)s has %(coins)d coins \  
worth a total of $%(amount).02f' % data  
  
print out
```

```
# Output: 'Tom has 2 coins worth a total of $2.40'
```

# Exceptions

```
>>> try:
...     1 / 0
... except:
...     print('That was silly!')
... finally:
...     print('This gets executed no matter what')
...
That was silly!
This gets executed no matter what
```

# File I/O

- **Reading file content:**

```
fd = open('filename.txt', 'r')
for line in fd:
    print line
fd.close()
```

```
open('filename.txt').read()
```

```
open('filename.txt').readlines()
```

- **Writing file content:**

```
fd open('filename.txt', 'w')
fd.write('Hello, World!')
fd.write('\n')
fd.close()
```



# What's next ?

## Documentation and pointers:

- <http://learnpythonthehardway.org/book/>
- <http://www.reddit.com/r/LearnPython> (**Ask your questions here**)
- <http://docs.python.org/2/>
- <http://reddit.com/r/Python> (**News**)
- <http://planet.python.org/> (**Blog aggregator**)
- <http://www.pythonweekly.com/> (**Newsletter**)



A tropical beach scene with a clear blue sky, white clouds, and turquoise water. Palm fronds are visible in the top and left corners.

**Thanks!**

<http://muthukadan.net>