**Introduction to Python**

Heavily based on presentations by
Matt Huenerfauth (Penn State)
Guido van Rossum (Google)
Richard P. Muller (Caltech)

-- Bhavani Shankar Ubuntu Developer

# Python

- Open source general-purpose language.
- Object Oriented, Procedural, Functional
- Easy to interface with C/ObjC/Java/Fortran
- Easy-ish to interface with C++ (via SWIG)
- Great interactive environment
  - Downloads: http://www.python.org
  - Documentation: http://www.python.org/doc/
  - Free book: http://www.diveintopython.net

# Versions Available

- Current Versions available if you want to test are:

  2.7.3 and 3.3.0 (Although many apps are not compatible with python 3 as yet)

# The Technical stuff

Installing & Running Python

# Binaries Available

- Python comes pre-installed with Mac OS X and Linux.
- Windows binaries from http://python.org/
- You might not have to do anything if you are running linux!

# The Python Interpreter

- Interactive interface:

  bhavani@bhavani-spagetti-monster:~$ python

  Python 2.7.3 (default, Sep 26 2012, 21:53:58)

  [GCC 4.7.2] on linux2

  Type "help", "copyright", "credits" or "license" for more information.

  >>>

# The python interpreter

- Python interpreter evaluates inputs:

  >>> 3*(7+3)

  30

- Python prompts with ‘>>>’.
- To exit python:

  cntrl+D or quit()

# Running Programs on *NIX

- Python programs can be run via the below command:

  *python filename.py*

  *You could make the *.py file executable and add thefollowing #!/usr/bin/python to the top to make it runnable. (Normally called as a “shebang”)*

# Batteries Included!

- Large collection of proven modules included in the standard distribution.

  http://docs.python.org/modindex.html

# The Basics

# A Code Sample

```
x = 34 - 23          # A comment.
y = “Hello”          # Another one.
Z = 3.45             # Assigning values
if z == 3.45 or y == “Hello”: #comparison
x = x + 1
y = y + “ World”  #concatenation
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 Datatypes

- 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”””*

# Whitespaces

- Whitespace 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 and class definitions.)

# A permutation example (though confusing for a start)

```
def perm(l):
        # Compute the list of all permutations of l
    if len(l) <= 1:
                return [l]
    r = []
    for i in range(len(l)):
             s = l[:i] + l[i+1:]
             p = perm(s)
             for x in p:
              r.append(l[i:i+1] + x)
    return r
```

# 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...
```

# Assignments

• 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 "<pyshell#16>", line 1, in -toplevel-
y
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 _ 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*

# Exercise -1

- You eat a nice meal of 2 masala dosa's and 2 stuffed parathas at a resturant known for the same. The pricing is as follows

  1 Dosa @ Rs 40

  1 stuffed paratha @ Rs 20

  tax @ 10%

  Write a program to calculate total cost

# Reference semantics in Python

# Understanding Reference Semantics

- Assignment manipulates references

— x = y **does not make a copy** of the object y references

— x = y makes x reference the object y references

- Very useful; but beware!
- Example:

```
>>> a = [1, 2, 3]  # a now references the list [1, 2, 3]
>>> b = a   # b now references what a references
>>> a.append(4)  # This changes list a references
>>> print b  # if we print what b references
[1, 2, 3, 4]  # tada! Surprise!
```

Why??

# Continued ...

- There is a lot going on when we type:

  x = 3

  First, an integer 3 is created and stored in memory

  A name x is created

  An reference to the memory location storing the 3 is then assigned to the name x

  So: When we say that the value of x is 3

  we mean that x now refers to the integer 3

# Continued ...

• The data 3 we created is of type integer. In Python, the datatypes integer, float, and string (and tuple) are “immutable.”

• This doesn’t mean we can’t change the value of x, i.e. change what x refers to ...

• For example, we could increment x:

```
>>> x = 3
>>> x = x + 1
>>> print x
4
```

# Continued ....

If we increment x, then what's really happening is:

1. The reference of name x is looked up.

*>>> x = x + 1*

2. The value at that reference is retrieved.

3. The 3+1 calculation occurs, producing a new data element 4 which is assigned to a fresh memory location with a new reference.

4. The name x is changed to point to this new reference.

5. The old data 3 is garbage collected if no name still refers to it.

# Continued

- So, for simple built-in datatypes (integers, floats, strings), assignment behaves as you would expect:

```
>>>x = 3  # 3 is created and assigned to x

>>>y = x  # y created refers to 3

>>>y = 4  # y ref is changed, changed to 4

>>>print x # no effect

3
```

# Continued ...

- For other data types (lists, dictionaries, user-defined types), assignment works differently.

  These datatypes are “mutable.”

  When we change these data, we do it in place.

  We don’t copy them into a new memory address each time.

  If we type y=x and then modify y, both x and y are changed.

# Sequence types:
# Tuples, Lists, and Strings

# 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 (“, ‘, or “““).

*>>> st = “Hello World”*

*>>> st = = ‘Hello World’*

*>>> st = “““This is a multi-line*

*that uses triple 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))*

# 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 substrings

```
>>> a = 'abcde'
>>> 'c' in a
True
>>> 'cd' in a
True
>>> 'ac' in a
False
```

Be careful: the in keyword is also used in the syntax of 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’*

# Exercise - 2

- Write a program using strings in python to find current date and time of the system.

  Hint: Use the datetime module

  from datetime import datetime and a function called datetime.now()

# Mutability:
# Tuples vs. Lists

# Tuples: Immutable

*>>> t = (23, “abc”, 4.56, (2,3), “def”)*

*>>> t[2] = 3.14*

*Traceback (most recent call last):*

*File "<pyshell#75>", line 1, in -toplevel-tu[2] = 3.14*

*TypeError: object doesn't 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*

*>>> li.append([10, 11, 12])*

*[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

# 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}
```

# Exercise -3

You are a supermarket owner and on a day lets say 100 people come to your supermarket and shop for bananas apples and oranges.

Write a program to compute total sales on a given day

Hint: Use price as a dictionary mapping like

*price {*

*banana =*

*orange =*

*apple =*

# Functions

# 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

- 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 may affect the caller

```
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
```

# Few points to note

- 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

# Exercise - 4

Now that we have seen about functions, write a function that calculates your day to day expenses while in college :)

Hint use def func(expenses) and assign values to your day to day expenses under different names/variables.

# Control of Flow

# Examples

```
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’.”

x = 3
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.”
```

# Exercise - 5

Write a function is_even that takes a number x as input and returns True if x is even and False otherwise.

# Modules

# 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*

# Classes and Objects

# Object Oriented Design

- A software item that contains variables and methods
  - Object Oriented Design focuses on
  - Encapsulation:

  —dividing the code into a public interface, and a private implementation of that interface
  - Polymorphism:

  —the ability to overload standard operators so that they have appropriate behavior based on their context
  - Inheritance:

  —the ability to create subclasses that contain specializations of their parents

# An example

```
class BankAccount(object):
    def __init__(self, initial_balance=0):
        self.balance = initial_balance
    def deposit(self, amount):
        self.balance += amount
    def withdraw(self, amount):
        self.balance -= amount
    def overdrawn(self):
        return self.balance < 0
my_account = BankAccount(15)
my_account.withdraw(5)
print my_account.balance
```

# Example - 2

```
class Customer(object):
"""Produces objects that represent customers."""
def __init__(self, customer_id):
self.customer_id = customer_id

def display_cart(self):
print "I'm a string that stands in for the contents of your shopping cart!"

class ReturningCustomer(Customer):
"""For customers of the repeat variety."""
def __init__(self, customer_id):
self.customer_id = customer_id

def display_order_history(self):
print "I'm a string that stands in for your order history!"

monty_python = ReturningCustomer("ID: 123")
monty_python.display_cart()
monty_python.display_order_history()
```

# Private Data

- In Python anything with two leading underscores is private
  __a, __my_variable
  - Anything with one leading underscore is semi-private, and you should feel guilty accessing this data directly.
  _b
  - Sometimes useful as an intermediate step to making data private

# File I/O, Strings, Exceptions...

# An example

```
>>> a = 1
>>> b = 2.4
>>> c = “Bhavi”
>>> '%s has %d coins worth a total of $%.02f' % (c, a, b)
'Bhavi has 1 coins worth a total of $2.40'
```

# Exercise -6

- Create a my_file.txt file and paste the below content

  “ This is a python class organised by fsmk with instructors”

  Write a python program to read the file (Hint: you can use readline() function) and determine the occurance of 'by' and 'with' words.

Questions??

Thanks!