TATA

TATA ELXSI
engineering creativity

Python Scripting

Presenter: Learning and Development Team

Agenda

- Introduction to Python
- Working with Python
- Operators
- Program Flow Constructs
- Built-in functions
- Salient features of Python
- interactive "shell"
- basic types: numbers, strings

Introduction

- Python is a programming language that lets you work more quickly and integrate your systems more effectively.
  - Python is an easy to learn,
  - very clear, readable syntax
  - exception-based error handling
  - very high level dynamic data types

- It has efficient high-level data structures and a simple but effective approach to object-oriented programming.
  - OOP is optional

Introduction

- Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for
  - scripting and
  - rapid application development in many areas on most platforms.

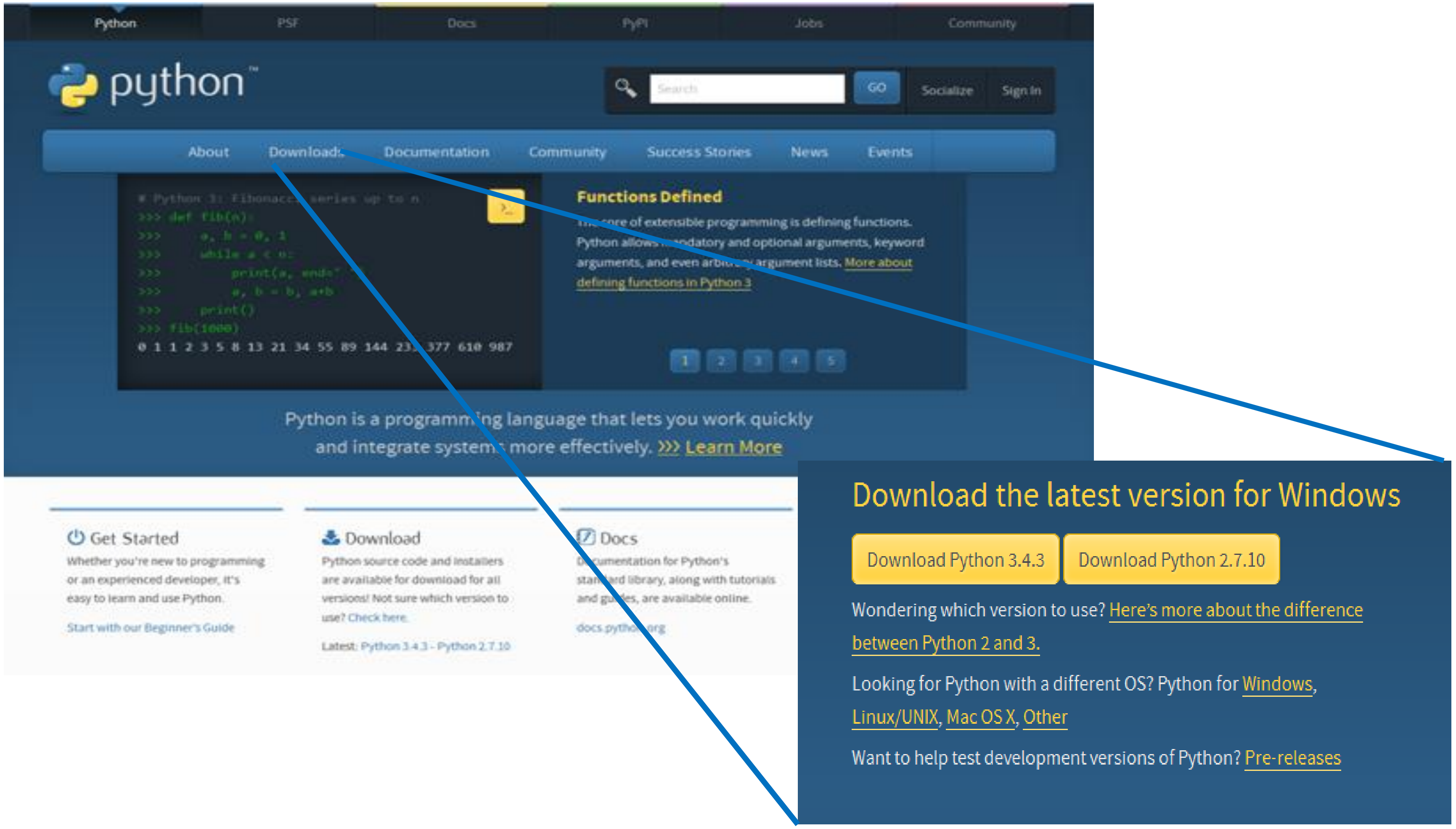
- Python can be integrated with another language.

- Python runs everywhere.

Where to get Python

- To download the appropriate file for your computer from

http://www.python.org/download



```
# Python 3: Fibonacci series up to n
>>> def fib(n):
>>>     a, b = 0, 1
>>>     while a < n:
>>>         print(a, end=' ')
>>>         a, b = b, a+b
>>>     print()
>>> fib(1000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

Functions Defined

The core of extensible programming is defining functions. Python allows mandatory and optional arguments, keyword arguments, and even arbitrary argument lists. More about defining functions in Python 3

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

Get Started

Whether you're new to programming or an experienced developer, it's easy to learn and use Python.

Start with our Beginner's Guide

Download

Python source code and installers are available for download for all versions! Not sure which version to use? Check here.

Latest: Python 3.4.3 - Python 2.7.10

Docs

Documentation for Python's standard library, along with tutorials and guides, are available online.

docs.python.org

Download the latest version for Windows

Download Python 3.4.3 | Download Python 2.7.10

Wondering which version to use? Here's more about the difference between Python 2 and 3.

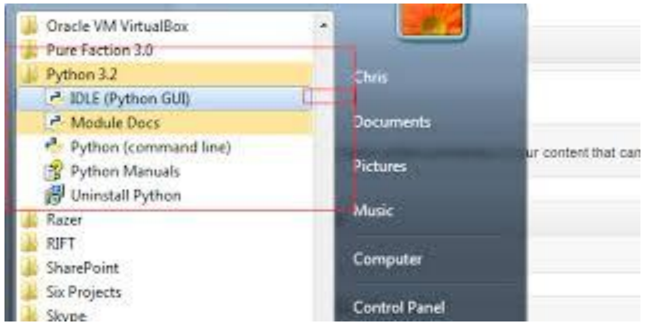
Looking for Python with a different OS? Python for Windows, Linux/UNIX, Mac OS X, Other

Want to help test development versions of Python? Pre-releases

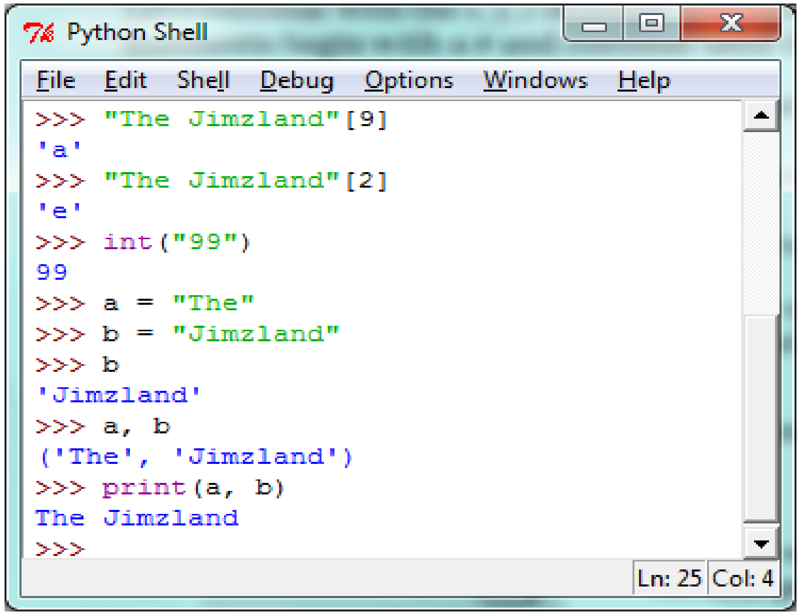
Interactive “Shell”

- Great for learning the language
- Great for experimenting with the library
- Great for testing your own modules
- Type statements or expressions at prompt:

```
>>> print "Hello, world"

Hello, world

>>> x = 12**2

>>> x/2

72

>>> # this is a comment
```



Two variations: IDLE (GUI),



```
>>> "The Jimzland"[9]
'a'
>>> "The Jimzland"[2]
'e'
>>> int("99")
99
>>> a = "The"
>>> b = "Jimzland"
>>> b
'Jimzland'
>>> a, b
('The', 'Jimzland')
>>> print(a, b)
The Jimzland
>>>
```

python (command line)

Creating and Running Programs

- Go into IDLE if you are not already.
- Go to File then New Window.

```
#!/usr/bin/python
print "Jack and Jill went up a hill"

print "to fetch a pail of water;"

print "Jack fell down, and broke his
    crown,"

print "and Jill came tumbling after.“
```

OutPut:

```
Jack and Jill went up a hill
to fetch a pail of water;
Jack fell down, and broke his
crown,
and Jill came tumbling after.
```

Basic operations for numbers

Operation	Symbol	Example
Exponentiation	**	5 ** 2 == 25
Multiplication	*	2 * 3 == 6
Division	/	14 / 3 == 4
Remainder	%	14 % 3 == 2
Addition	+	1 + 2 == 3
Subtraction	-	4 - 3 == 1

Using Python from the command line

```
$python

Python 3.3.2 (v3.3.2:d047928ae3f6, May 16 2013, 00:03:43) [MSC v.1600 32 bit (Intel)]
on win32

Type "copyright", "credits" or "license()" for more information.

>>>  print "2 + 2 is", 2+2

        2 + 2 is 4

>>>print "3 * 4 is", 3 * 4

        3 * 4 is 12

>>>print 100 - 1, " = 100 - 1"

        99 = 100 – 1

>>>print "(33 + 2) / 5 + 11.5 = ",(33 + 2) / 5 + 11.5

        (33 + 2) / 5 + 11.5 = 18.5
```

```
#!/usr/bin/python

print "2 + 2 is", 2+2

print "3 * 4 is", 3 * 4

print 100 - 1, " = 100 - 1"

print "(33 + 2) / 5 + 11.5 = ",(33 + 2) / 5 + 11.5
```

The output when the program is run:

```
2 + 2 is 4

3 * 4 is 12

99 = 100 - 1

(33 + 2) / 5 + 11.5 = 18.5
```

Numbers

- The usual suspects
  - 12, 3.14, 0xFF, 0377, (-1+2)*3/4**5, abs(x), 0<x<=5
- C-style shifting & masking
  - 1<<16, x&0xff, x|1, ~x, x^y
- Integer division truncates :-(
  - 1/2 -> 0 # 1./2. -> 0.5, float(1)/2 -> 0.5
  - Will be fixed in the future
- Long (arbitrary precision), complex
  - 2L**100 -> 1267650600228229401496703205376L
    - In Python 2.2 and beyond, 2**100 does the same thing
  - 1j**2 -> (-1+0j)

precedence and associativity

- The order of operations is the same as in math:

- 1. parentheses ()
- 2. exponents **
- 3. multiplication *, division \, and remainder %
- 4. addition + and subtraction -

Strings

"hello"+"world"	"helloworld"	# concatenation
"hello"*3	"hellohellohello"	# repetition
"hello"[0]	"h"	# indexing
"hello"[-1]	"o"	# (from end)
"hello"[1:4]	"ell"	# slicing
len("hello")	5	# size
"hello" < "jello"	1	# comparison
"e" in "hello“	1	# search

- "escapes: \n etc, \033 etc, \if etc"
- 'single quotes' """triple quotes""" r"raw strings"

Floating point numbers

- print "14 / 3 = ",14 / 3
- print "14 % 3 = ",14 % 3
- print
- print "14.0 / 3.0 =",14.0 / 3.0
- print "14.0 % 3.0 =",14 % 3.0
- print
- print "14.0 / 3 =",14.0 / 3
- print "14.0 % 3 =",14.0 % 3
- print
- print "14 / 3.0 =",14 / 3.0
- print "14 % 3.0 =",14 % 3.0
- print

input/output operations

```
#!/usr/bin/python
#Author:
#Date:
#purpose : Input / Output and Variables
num = input("Type in a Number: ")
str = raw_input("Type in a String: ")
print "num =", num
print "num is a ",type(num)
print "num * 2 =",num*2
print "str =", str
print "str is a ",type(str)
print "str * 2 =",str*2
```

The output:

```
Type in a Number: 12.34
Type in a String: Hello
num = 12.34
num is a <type ’float’>
num * 2 = 24.68
str = Hello
str is a <type ’string’>
str * 2 = HelloHello
```

Relational Operators

Operator	function
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equal
!=	not equal
<>	another way to say not equal

Program Flow Constructs

Agenda

- control structures
- if Statements
- for Statements
- The range() Function
- break and continue Statements
- else Clauses on Loops
- pass Statements
- functions & procedures

Control Structures

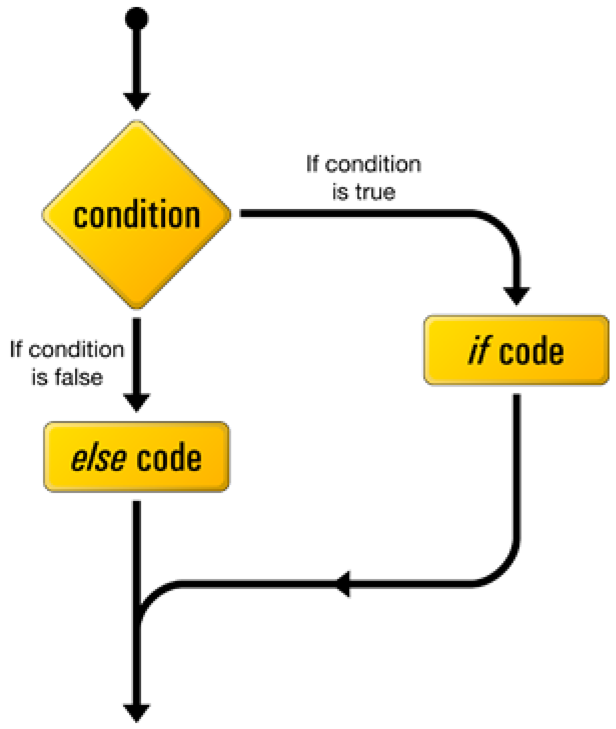
```
if condition:
    statements
[elif condition:
    statements] ...
else:
    statements
```

```
while condition:
    statements

for var in sequence:
    statements

break
continue
```

TATA ELXSI engineering creativity

if Statements

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
...
More
```



if Statements

```
#!/usr/bin/python
#Date:
#Purpose :  Plays the guessing game higher or lower

number = 78
guess = 0
while guess != number :
    guess = input ("Guess a number: ")
    if guess > number :
        print "Too high"
    elif guess < number :
        print "Too low“

    print "Just right"
```

The for loop

- **The range() Function**

range(5, 10) 5 through 9

range(0, 10, 3) 0, 3, 6, 9

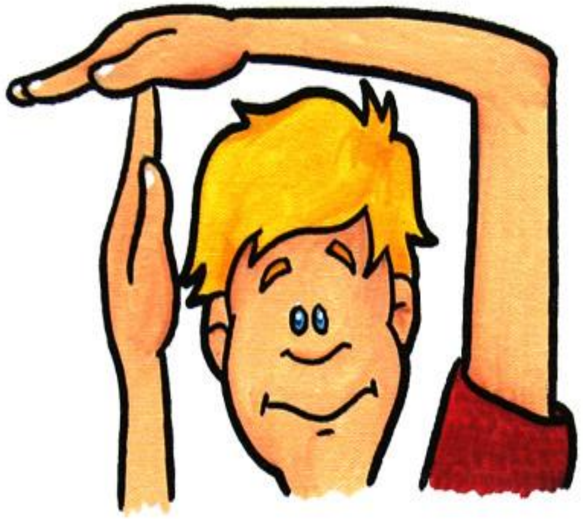
range(-10, -100, -30) -10, -40, -70

```
>>> for i in range(5):
 ...    print(i)
...
```

Break statement

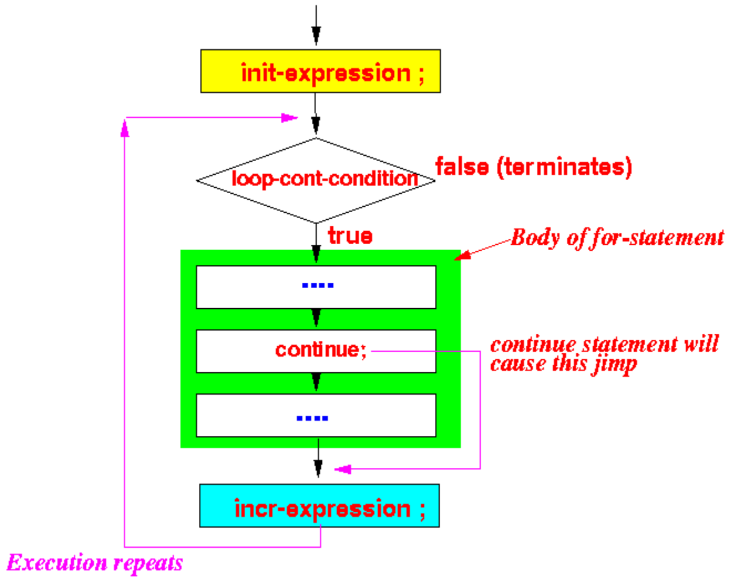
The break statement, like in C, breaks out of the smallest enclosing for or while loop.

```
for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'equals', x, '*', n//x)
...             break
...     else:
... # loop fell through without finding a factor
...         print(n, 'is a prime number')
...
```



Continue statement

The continue statement, also borrowed from C, continues with the next iteration of the loop:



```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print("Found an even number", num)
...         continue
...     print("Found a number", num)
```

Else and loop

- Loop statements may have an else clause;
- It is executed when the loop terminates through exhaustion of the list (with for) or when the condition becomes false (with while).
- But not when the loop is terminated by a break statement.

Pass Statements

- The pass statement does nothing.
- It can be used when a statement is required syntactically but the program requires no action.

```
>>> while True:
... pass # Busy-wait for keyboard interrupt (Ctrl+C) ...
```

This is commonly used for creating minimal classes:

```
>>> class MyEmptyClass:
... pass
...
```

Grouping Indentation

In Python:

```
for i in range(20):
    if i%3 == 0:
        print i
        if i%5 == 0:
            print "Bingo!"
    print "---"
```

In C:

```
for (i = 0; i < 20; i++)
{
    if (i%3 == 0) {
        printf("%d\n", i);
        if (i%5 == 0) {
            printf("Bingo!\n"); }
      }
      printf("---\n");
}
```

```
0
Bingo!
---
---
---
3
---
---
---
6
---
---
---
9
---
---
---
12
---
---
---
15
Bingo!
---
---
---
18
---
---
```

Functions, Procedures

```
def name(arg1, arg2, ...):
    """documentation"""# optional doc string
    statements

return                      # from procedure
return expression           # from function
```

Example Function

```
def gcd(a, b):
    "greatest common divisor"
    while a != 0:
        a, b = b%a, a    # parallel assignment
    return b

>>> gcd.__doc__
'greatest common divisor'
>>> gcd(12, 20)
4
```

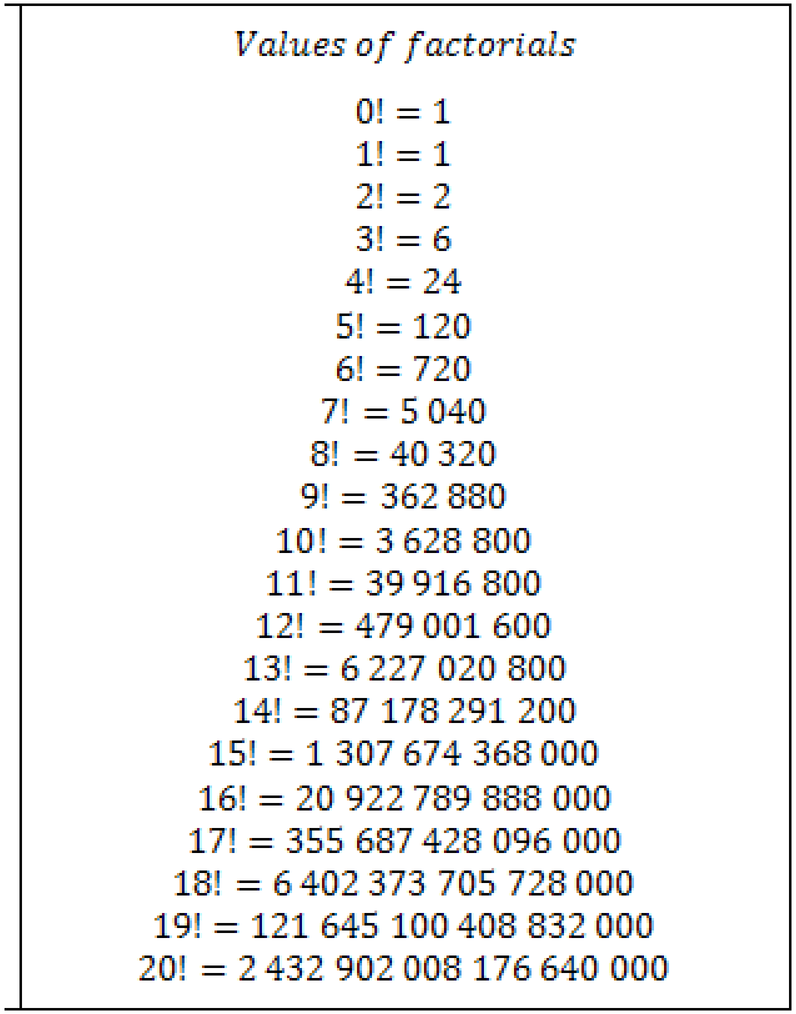
Example Function

```
a_var = 10
b_var = 15
e_var = 25
def a_func(a_var):
        print "in a_func a_var = ",a_var
        b_var = 100 + a_var
        d_var = 2*a_var
        print "in a_func b_var = ",b_var
        print "in a_func d_var = ",d_var
        print "in a_func e_var = ",e_var
        return b_var + 10
```

```
c_var = a_func(b_var)
print "a_var = ",a_var
print "b_var = ",b_var
print "c_var = ",c_var
print "d_var = ",d_var
```

Example Function

```
#defines a function that calculates the factorial
def factorial(n):
        if n <= 1:
                return 1
        return n*factorial(n-1)

print "2! = ",factorial(2)
print "3! = ",factorial(3)
print "4! = ",factorial(4)
print "5! = ",factorial(5)
```



Values of factorials

0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5 040
8! = 40 320
9! = 362 880
10! = 3 628 800
11! = 39 916 800
12! = 479 001 600
13! = 6 227 020 800
14! = 87 178 291 200
15! = 1 307 674 368 000
16! = 20 922 789 888 000
17! = 355 687 428 096 000
18! = 6 402 373 705 728 000
19! = 121 645 100 408 832 000
20! = 2 432 902 008 176 640 000

Data Structures

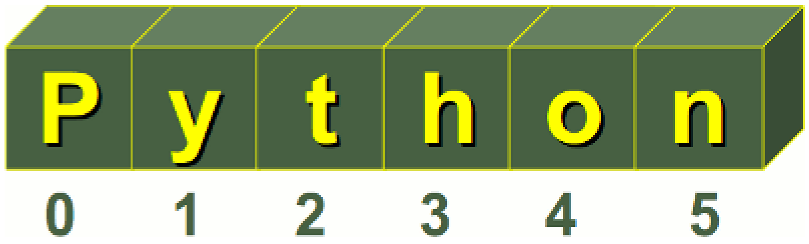
Agenda

- Sequences
  - Strings
  - Lists
  - Tuples
    - Membership
    - Concatenation
    - Repetition
    - Slices
    - Conversions
    - Built-in functions
  - Implementing Data Structures like Stack and Queue

strings

- Strings are amongst the most popular types in Python.
- We can create them simply by enclosing characters in quotes.
- Python treats single quotes the same as double quotes.

```
>>> a='vikas'
>>> b="karanth"
>>> a+b
'vikaskaranth'
>>> print(a+b)
Vikaskaranth
>>> print(a,b)
vikas karanth
>>>
```



strings

```
#!/usr/bin/python

var1 = 'Hello World!'
var2 = "Python Programming"

print "var1[0]: ", var1[0]
print "var2[1:5]: ", var2[1:5]
```

Output:

```
var1[0]: H
var2[1:5]: ytho
```

```
#!/usr/bin/python
var1 = 'Hello World!'
print "Updated String :- ", var1[:6] +
    'Python‘
```

Output:

```
Updated String :- Hello Python
```

Operator	Description	Example
+	Concatenation - Adds values on either side of the operator	a + b will give HelloPython
*	Repetition - Creates new strings, concatenating multiple copies of the same string	a*2 will give -HelloHello
[]	Slice - Gives the character from the given index	a[1] will give **e**
[:]	Range Slice - Gives the characters from the given range	a[1:4] will give **ell**
in	Membership - Returns true if a character exists in the given string	**H in a** will give 1
not in	Membership - Returns true if a character does not exist in the given string	**M not in a** will give 1
r/R	Raw String - Suppresses actual meaning of Escape characters. The syntax for raw strings is exactly the same as for normal strings with the exception of the raw string operator, the letter "r," which precedes the quotation marks. The "r" can be lowercase (r) or uppercase (R) and must be placed immediately preceding the first quote mark.	**print r'\n'** prints \n and **print R'\n'** prints \n
%	Format - Performs String formatting	See at next section

strings

```
>>> 'spam eggs'  # single quotes
'spam eggs'
>>> 'doesn\'t'  # use \' to escape the single
      quote...
"doesn't"
>>> "doesn't"  # ...or use double quotes instead
"doesn't"
>>> '"Yes," he said.'
'"Yes," he said.'
>>> "\"Yes,\" he said."
'"Yes," he said.'
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
```

```
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'

>>> print('"Isn\'t," she said.')
"Isn't," she said.

>>> s = 'First line.\nSecond line.'  # \n
means newline

>>> s  # without print(), \n is included in
the output
'First line.\nSecond line.'
>>> print(s)  # with print(), \n produces a
new line
First line.
Second line.
```

strings

- Strings can be indexed (subscripted), with the first character having index 0.

```
>>> word = 'Python'
>>> word[0]  # character in position 0
'P'
>>> word[5]  # character in position 5
'n'

>>> word[-1]  # last character
'n'
>>> word[-2]  # second-last character
'o'
>>> word[-6]
'P'
```

```
>>> word[0:2]  # characters from position 0 (included)
        to 2 (excluded)
'Py'
>>> word[2:5]  # characters from position 2 (included)
        to 5 (excluded)
'tho‘

>>> word[:2] + word[2:]
'Python'
>>> word[:4] + word[4:]
'Python'
```

strings

```
>>> word[:2]  # character from the beginning to position 2 (excluded)
'Py'
>>> word[4:]  # characters from position 4 (included) to the end
'on'
>>> word[-2:] # characters from the second-last (included) to the end
'on'
```

```
 +---+---+---+---+---+---+
 | P | y | t | h | o | n |
 +---+---+---+---+---+---+
 0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

String Formatting Operator:

- One of Python's coolest features is the string format operator **%.**

```
#!/usr/bin/python

>>print "My name is %s and weight is %d kg!" % ('Zara', 21)
```

String Formatting Operator:

Format Symbol	Conversion
%c	character
%s	string conversion via str() prior to formatting
%i	signed decimal integer
%d	signed decimal integer
%u	unsigned decimal integer
%o	octal integer
%x	hexadecimal integer (lowercase letters)
%X	hexadecimal integer (UPPERcase letters)
%e	exponential notation (with lowercase 'e')
%E	exponential notation (with UPPERcase 'E')
%f	floating point real number
%g	the shorter of %f and %e
%G	the shorter of %f and %E

List in Python

Python Lists:

- The list is a most versatile data type available in Python which can be written as a list of comma-separated values (items) between square brackets.

- Good thing about a list is that items in a list need not all have the same type.

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

>>>  print "list1[0]: ", list1[0]
>>>  print "list2[1:5]: ", list2[1:5]
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> letters
['a', 'b', 'c', 'd', 'e', 'f', 'g']

>>> # replace some values
>>> letters[2:5] = ['C', 'D', 'E']

>>> letters
['a', 'b', 'C', 'D', 'E', 'f', 'g']
```

```
>>> # now remove them
>>> letters[2:5] = []
>>> letters
['a', 'b', 'f', 'g']

>>> # clear the list by replacing all the
    elements with an empty list
>>> letters[:] = []
>>> letters
[]
```

Lists

- Flexible arrays, *not* Lisp-like linked lists
  - a = [99, "bottles of beer", ["on", "the", "wall"]]
- Same operators as for strings
  - a+b, a*3, a[0], a[-1], a[1:], len(a)
- Item and slice assignment
  - a[0] = 98
  - a[1:2] = ["bottles", "of", "beer"]

-> [98, "bottles", "of", "beer", ["on", "the", "wall"]]

  - del a[-1]   # -> [98, "bottles", "of", "beer"]

More List Operations

```
>>> a = range(5)                 # [0,1,2,3,4]
>>> a.append(5)                  # [0,1,2,3,4,5]
>>> a.pop()            # [0,1,2,3,4]
5
>>> a.insert(0, 42)    # [42,0,1,2,3,4]
>>> a.pop(0)           # [0,1,2,3,4]
5.5
>>> a.reverse()                  # [4,3,2,1,0]
>>> a.sort()           # [0,1,2,3,4]
```

Lists : example

```
which_one = input("What month (1-12)? ")
months = [’January’, ’February’, ’March’, ’April’, ’May’, ’June’, \ ’July’, ’August’,
’September’, ’October’, ’November’, \ ’December’]
if 1 <= which_one <= 12:
        print "The month is",months[which_one - 1]
```

Using Lists as Stacks

- To add an item to the top of the stack, use append().
- To retrieve an item from the top of the stack, use pop() without an explicit index.
- Exercise : Implement stack using List

Using Lists as Queues

- To implement a queue pop(0)

Exercise: Implement queue using List

Dictionaries

- Dictionaries consist of pairs (called items) of keys and their corresponding values.

- Hash tables, "associative arrays“

```
>>> dict = {'Alice': '2341', 'Beth': '9102', 'Cecil': '3258'}
```

- Lookup:

```
>>>dict[“Beth"] -> “3452"

>>>dict["back"] # raises KeyError exception
```

Delete Dictionary Elements:

- You can either remove individual dictionary elements or clear the entire contents of a dictionary.
- You can also delete entire dictionary in a single operation.

```
#!/usr/bin/python
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}

del dict['Name']; # remove entry with key 'Name'
dict.clear(); # remove all entries in dict
del dict ; # delete entire dictionary

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

More Dictionary Ops

- Keys, values, items:
  - d.keys() -> ["duck", "back"]
  - d.values() -> ["duik", "rug"]
  - d.items() -> [("duck","duik"), ("back","rug")]
- Presence check:
  - d.has_key("duck") -> 1; d.has_key("spam") -> 0
- Values of any type; keys almost any
  - {"name":"Guido", "age":43, ("hello","world"):1, 42:"yes", "flag": ["red","white","blue"]}

Dictionary Details

- Keys must be **immutable**:
  - numbers, strings, tuples of immutables
    - these cannot be changed after creation
  - reason is *hashing* (fast lookup technique)
  - **not** lists or other dictionaries
    - these types of objects can be changed "in place"
  - no restrictions on values
- Keys will be listed in **arbitrary order**
  - again, because of hashing

Tuples

- key = (lastname, firstname)
- point = x, y, z   # parentheses optional
- x, y, z = point   # unpack
- lastname = key[0]
- singleton = (1,)   # trailing comma!!!
- empty = ()   # parentheses!
- tuples vs. lists; tuples immutable

Variables

- No need to declare
- Need to assign (initialize)
    - use of uninitialized variable raises exception
- Not typed

```
if friendly: greeting = "hello world"

else: greeting = 12**2

print greeting
```

- ***Everything*** is a "variable":
    - Even functions, classes, modules

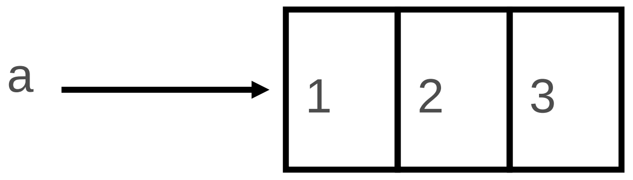
Reference Semantics

- Assignment manipulates references
  - x = y **does not make a copy** of y
  - x = y makes x **reference** the object y references
- Very useful; but beware!
- Example:

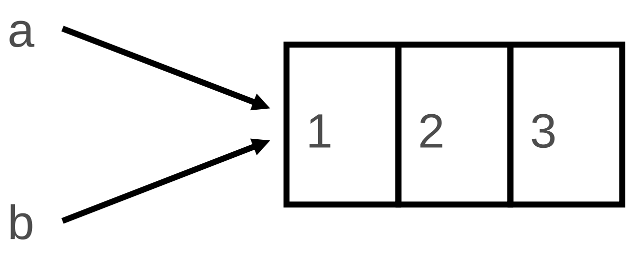
```
>>> a = [1, 2, 3]

>>> b = a

>>> a.append(4)

>>> print b

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

Changing a Shared List

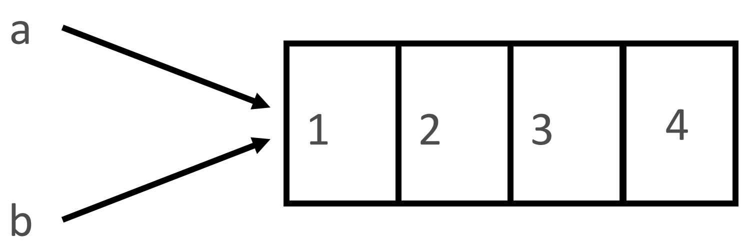
```
a = [1, 2, 3]
```



```
b = a
```



```
a.append(4)
```

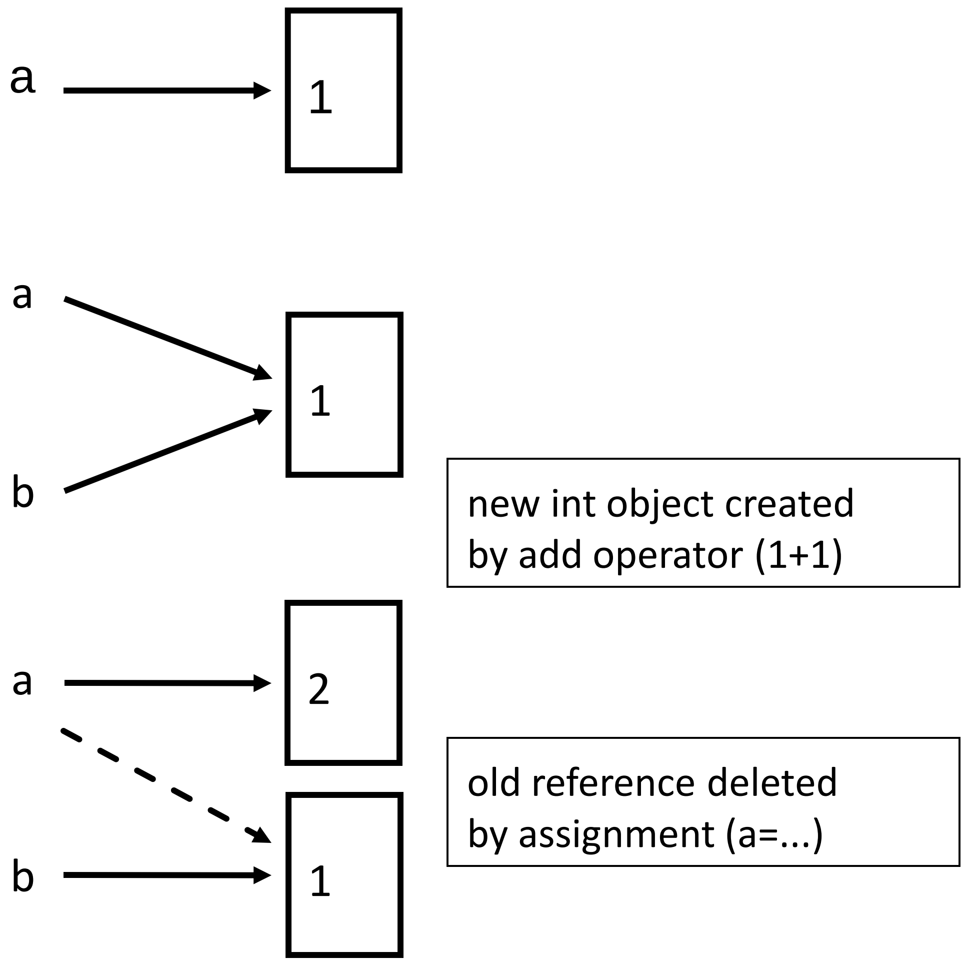


Changing an Integer

a = 1

b = a

a = a+1


a
1
a
b
1
new int object created by add operator (1+1)
a
2
b
1
old reference deleted by assignment (a=...)

Modules

Agenda

- Why use modules?
- Using the os, sys modules
- How import works
- Different ways of importing
- Making your own Modules
- Creating your own Modules
- Module Namespaces
- Changing the module search path

What is module?

- A module allows you to logically organize your Python code.
- Grouping related code into a module makes the code easier to understand and use.
- A module is a Python object with arbitrarily named attributes that you can bind and reference.
- Simply, a module is a file consisting of Python code.
- A module can define functions, classes and variables.
- A module can also include run able code.

Module

- Example:

```
def print_func( par ):
    print ("Hello : ", par)
    return
print_func("world")
```

The import Statement:

- You can use any Python source file as a module by executing an import statement in some other Python source file.
- The import has the following syntax:
  - import module1[, module2[,... moduleN]
- When the interpreter encounters an import statement, it imports the module if the module is present in the search path.
- A search path is a list of directories that the interpreter searches before importing a module.

Import statement

- For example, to import the module hello.py, you need to put the following command at the top of the script:

- #!/usr/bin/python

- # Import module support
- import support

- # Now you can call defined function that module as follows
- support.print_func("Zara")

Import Example

Example2:

```
# Fibonacci numbers module


def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print(b, end=' ')
        a, b = b, a+b
    print()
```

```
# return Fibonacci series up to n

def fib2(n):
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

TATA ELXSI engineering creativity

Import Example

- 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
        OR
>>> from fibo import *
>>> fib(500)
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.

Standard Modules

- Python comes with a library of standard modules
- Some modules are built into the interpreter
- provide access to operating system primitives such as system calls.
- One particular module deserves some attention: sys , which is built into every Python interpreter.
- The variables sys.ps1 and sys.ps2 define the strings used as primary and secondary prompts:
- >>> import sys
- >>> sys.ps1
- '>>> '
- >>> sys.ps2
- '... '

Standard Modules

- The built-in function **dir()** is used to find out which names a module defines.
- It returns a sorted list of strings:

- >>> import sys
- >>> dir(sys)

['__displayhook__', '__doc__', '__excepthook__', '__loader__', '__name__',
 '__package__', '__stderr__', '__stdin__', '__stdout__',
 '_clear_type_cache', '_current_frames', '_debugmallocstats', '_getframe',
 '_home', '_mercurial', '

_xoptions', 'api_vers...ion', 'argv',..........

Files

Agenda

•Open – close the files
•Reading files
•Writing to the files

Files

- You can open and use files for reading or writing by creating an object of the file class and using its read, readline or write methods appropriately to read from or write to the file.
- The ability to read or write to the file depends on the mode you have specified for the file opening.
- The close method to tell Python that we are done using the file.

Reading and Writing Files

- **open()** returns a ***file object***, and is most commonly used with two arguments: open(filename, mode).
- >>> f = open('workfile', 'w')
- Normally, files are opened in *text mode.*
- To read a file's contents, call f.read(size), which reads some quantity of data and returns it as a string or bytes object.
- *size* is an optional numeric argument.
- When *size* is omitted or negative, the entire contents of the file will be read and returned;

Reading and Writing Files

- f.readline() reads a single line from the file;
- >>> f.readline()
  - returns an empty string, the end of the file has been reached, while a blank line is represented by '\n', a string containing only a single newline.

- For reading lines from a file, you can loop over the file object.

```
>>> for line in f:
... print(line, end='')
...
This is the first line of the file.
Second line of the file
```

Reading and Writing Files

- f.write(string) writes the contents of *string* to the file, returning the number of characters written.

```
>>> f.write('This is a test\n')
15
```

- To write something other than a string, it needs to be converted to a string first:

```
>>> value = ('the answer', 42)
>>> s = str(value)
>>> f.write(s)
18
```

Tell

- f.tell() returns an integer giving the file object’s current position in the file, measured in bytes from the beginning of the file.

- To change the file object’s position, use f.seek(offset, from_what).

A *from_what* value of 0 measures from the beginning of the file

1 uses the current file position, and

2 uses the end of the file as the reference point.

Errors and Exceptions

Agenda

- Errors
- Try..Except
- Handling Exceptions
- Raising Exceptions
- How To Raise Exceptions
- Try..Finally
- Using Finally

Errors and Exceptions

- **Syntax Errors :** Syntax errors, also known as parsing errors

```
>>> while True
print('Hello world')

File "<stdin>", line 1, in ?

while True print('Hello world')

                    ^

 SyntaxError: invalid syntax
```

Exceptions

Errors detected during execution are called *exceptions*

```
>>> while True:
...     try:
...         x = int(input("Please enter a number: "))
...         break
...     except ValueError:
...         print("Oops!  That was no valid number.  Try again...")
```

Exceptions

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: int division or modulo by zero

>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'spam' is not defined

>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: Can't convert 'int' object to str implicitly
```

Handling Exceptions

- It is possible to write programs that handle selected exceptions.

```
>>> while True :
...     try :
...             x =int(input("Enter a number"))
...             break
...     except ValueError :
...             print ("OOPS error")

Enter a number
OOPS error
Enter a number
OOPS error
```

Try statement

- The **try** statement works as follows.
    - First, the try *clause* (the statement(s) between the **try** and **‘except’** keywords) is executed.
    - If no exception occurs, the *except clause* is skipped and execution of the **try** statement is finished.
    - If an exception occurs during execution of the try clause, the rest of the clause is skipped.
        - Then if its type matches the exception named after the **except** keyword, the except clause is executed, and then execution continues after the **try** statement.
    - If an exception occurs which does not match the exception named in the except clause, it is passed on to outer ‘**try’** statements;
    - if no handler is found, it is an *unhandled exception* and execution stops

Try statement

- A try statement may have more than one except clause, to specify handlers for different exceptions.
- Handlers only handle exceptions that occur in the corresponding try clause, not in other handlers of the same try statement.
- An except clause may name multiple exceptions as a parenthesized tuple, for example:
- ... except (RuntimeError, TypeError, NameError):
- ...   pass

Example

```
import sys
try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except IOError as err:
    print("I/O error: {0}".format(err))
except ValueError:
    print("Could not convert data to an integer.")
except:
    print("Unexpected error:", sys.exc_info()[0])
    raise
```

```
I/O error: [Errno 2] No such file or directory: 'myfile.txt'
>>> ================================
Could not convert data to an integer.
```

else clause - Example

- The try ... except statement has an optional else clause, which, when present, must follow all except clauses.

- It is useful for code that must be executed if the try clause does not raise an exception.

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print('cannot open', arg)
    else:
        print(arg, 'has', len(f.readlines()), 'lines')
        f.close()
```

- The use of the **‘else’** clause is better than adding additional code to the **‘try’** clause.

- it avoids accidentally catching an exception that wasn’t raised by the code being protected by the **try ... except** statement.

- Exception handlers don’t just handle exceptions if they occur immediately in the try clause, but also if they occur inside functions that are called (even indirectly) in the try clause.

```
>>> def this_fails():
...     x = 1/0
...
>>> try:
...     this_fails()
... except ZeroDivisionError as err:
...     print('Handling run-time error:', err)
...

Handling run-time error: int division or modulo by zero
```

Raising Exceptions

- The raise statement allows the programmer to force a specified exception to occur.

```
>>> raise NameError('HiThere')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: HiThere
```

- The sole argument to ‘**raise**’ indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from ‘**Exception**’).

Re-Raising Exceptions

- If you need to determine whether an exception was raised but don’t intend to handle it,
- A simpler form of the raise statement allows you to re-raise the exception:

```
>>> try:
...     raise NameError('HiThere')
... except NameError:
...     print('An exception flew by!')
...     raise
...
An exception flew by!
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
NameError: HiThere
```

User-defined Exceptions

- Programs may name their own exceptions by creating a new exception class.

- Exceptions should typically be derived from the Exception class, either directly or indirectly.

User-defined Exceptions

```
>>> class MyError(Exception):
...     def __init__(self, value):
...         self.value = value
...     def __str__(self):
...         return repr(self.value)
...
>>> try:
...     raise MyError(2*2)
... except MyError as e:
...     print('My exception occurred, value:', e.value)
... finally:
...     print(‘surely this is executed’)
...
My exception occurred, value: 4
```

Object-Oriented Programming

Class and Objects

Agenda

- Introduction
- Creating a Class
    - Class Objects
    - Instance Objects
    - Method Objects
- Methods
- “self” the self reference
- Object Methods
- The __init__ method
- Class and Object Variables

Classes

- Python classes provide all the standard features of Object Oriented Programming:
- the class inheritance mechanism allows multiple base classes,
- a derived class can override any methods of its base class or classes, and
- a method can call the method of a base class with the same name.

Class Definition Syntax

- In C++ terminology, normally class members (including the data members) are *public* and all member functions are *virtual*.

- The simplest form of class definition looks like this:

- class ClassName:

  <statement-1>

  . . .

  <statement-N>

- Class definitions, like function definitions (**‘def’** statements) must be executed before they have any effect.

Defining class – class Objects

Class objects support two kinds of operations: attribute references and instantiation.

```
class MyClass:
    """A simple example class"""
    i = 12345
    def f(self):
        return 'hello world'
```

- Class *instantiation* uses function notation.

```
x = MyClass()
```

- creates a new *instance* of the class and assigns this object to the local variable x

Class Objects

- The instantiation operation (“calling” a class object) creates an empty object.
- Many classes like to create objects with instances customized to a specific initial state.
- Therefore a class may define a special method named __init__().
- class instantiation automatically invokes ‘**__init__()**’ for the newly-created class instance.

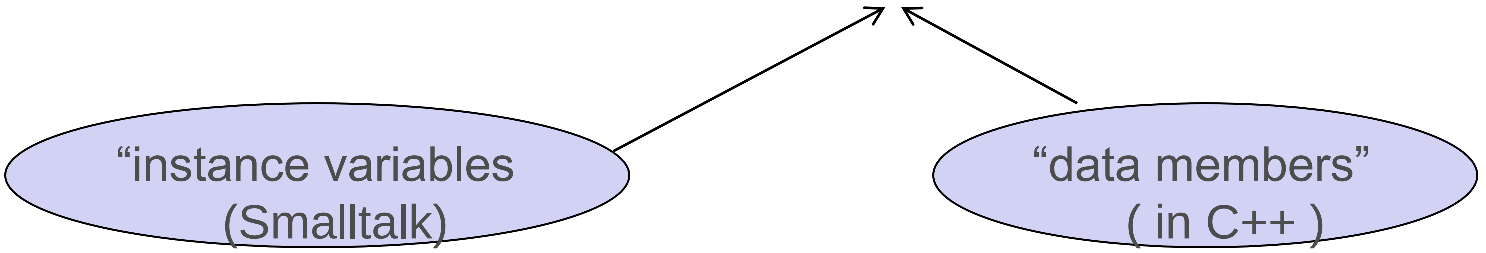
Class Objects

- class instantiation operator are passed on to __init__(). For example,

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

Instance Objects

- The only operations understood by instance objects are attribute references.
- There are two kinds of valid attribute names, data attributes and methods.



- Data attributes need not be declared; like local variables, they spring into existence when they are first assigned to.

```
x.counter = 1
while x.counter < 10:
        x.counter = x.counter * 2
print(x.counter)
del x.counter
```

- A method is a function that “belongs to” an object.

Method Objects

- Usually, a method is called right after it is bound:
- x.f()   # In the MyClass example, this will return the string 'hello world'.
- it is not necessary to call a method right away: x.f is a method object.
- It can be stored away and called at a later time.

```
xf = x.f
while True:
    print(xf())
```

- will continue to print hello world until the end of time.

Instances of an "empty" class like SomeName have no behavior, but they may have state.

```
class Behave(object):
    def _ _init_ _(self, name):
        self.name = name
    def once(self):
        print "Hello,", self.name
    def rename(self, newName):
        self.name = newName
    def repeat(self, N):
        for i in range(N): self.once( )
```

```
beehive = Behave("Queen Bee")
beehive.repeat(3)

beehive.rename("Stinger")
beehive.once( )

print (beehive.name)
beehive.name = 'See, you can rebind
it "from the outside" too, if you
want'


beehive.repeat(2)
```

TATA ELXSI engineering creativity

Object-Oriented Programming

Inheritance

Agenda

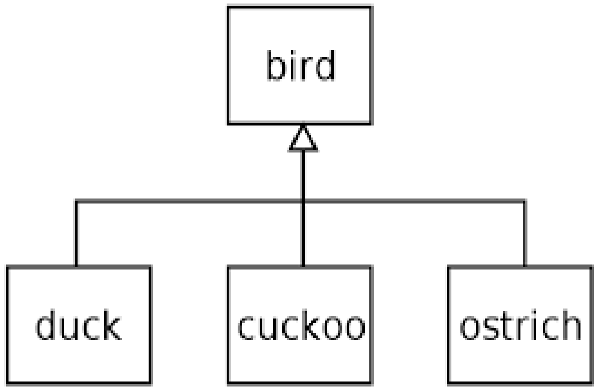
- Inheritance
- Multiple-inheritance
- Operator Overloading

inheritance

- Python supports inheritance

- The syntax for a derived class definition looks like this:

```
class DerivedClassName(BaseClassName):
    <statement-1>
    . . .
    <statement-N>
```



The name BaseClassName must be defined in a scope containing the derived class definition.

- This can be useful, for example, when the base class is defined in another module:
- class DerivedClassName(modname.BaseClassName):

inheritance

- if a requested attribute is not found in the derived class, the search proceeds to look in the base class.
- This rule is applied recursively if the base class itself is derived from some other class.
- Derived classes may override methods of their base classes.
- An overriding method in a derived class may in fact want to extend rather than simply replace the base class method of the same name.
- There is a simple way to call the base class method directly: just call BaseClassName.methodname(self, arguments).

inheritance

- Python has two built-in functions that work with inheritance:
- Use isinstance() to check an instance's type: isinstance(obj, int) will be True only if obj.__class__ is int or some class derived from int .
- Use issubclass() to check class inheritance: issubclass(bool, int) is True since bool is a subclass of int . However, issubclass(unicode, str) is False since unicode is not a subclass of str   (they only share a common ancestor, basestring).

Inheritance: example

```
class Topbase(object):
    def __init__(self):
        print("In TopBase")
        self.i=100
    def Display(self):
        print("In TopBase value is : ",self.i)

class derivedOne(Topbase):
    def __init__(self):
        print("In DerivedOne")
        self.j=50
    def Display(self):
        print("In derivedOne value is : ",self.j)
```

Inheritance: example

```
class derivedTwo(derivedOne):
    def __init__(self):
        print("In DerivedOne")
        self.k=25
    def Display(self):
        print("In derivedTwo value is : ",self.k)

oderTwo=derivedTwo()
#oderTwo.Display()
print derivedTwo.__mro__
```

Output:

```
>>>
In DerivedOne
(<class '__main__.derivedTwo'>, <class '__main__.derivedOne'>, <class
'__main__.Topbase'>, <type 'object'>)
>>>
```

Calling init in derived class: example

```
class Car(object):
    condition = "new"

    def __init__(self, model, color, mpg):
        self.model = model
        self.color = color
        self.mpg   = mpg
    def Display(self):
        print(condition)
        print(self.model)
        print(self.color)
        print(self.mpg)
```

Calling init in derived class: example

```
class ElectricCar(Car):
    def __init__(self, battery_type, model, color, mpg):
        self.type=battery_type
        super(ElectricCar, self).__init__(model, color, mpg)
    def Display(self):
        print(Car.condition)
        print(self.model)
        print(self.color)
        print(self.mpg)
        print(self.type)


myCar=ElectricCar("battery",2015,"silver",25)
myCar.Display()
```

Multi-level inheritance

```
class Topbase(object):
    def __init__(self,ii):
        print("In TopBase")
        self.i=ii
    def Display(self):
        print("In TopBase value is : ",self.i)

class derivedOne(Topbase):
    def __init__(self,ii,jj):
        print("In DerivedOne")
        self.j=jj
        super(derivedOne,self).__init__(ii)
    def Display(self):
        print("In derivedOne value is : ",self.j)
```

TATA ELXSI engineering creativity

Multi-level inheritance

```
class derivedTwo(derivedOne):
    def __init__(self,ii,jj,kk):
        print("In DerivedOne")
        self.k=kk
        super(derivedTwo, self).__init__(ii,jj)

    def Display(self):
        print("In derivedTwo value is : ",self.k)
        print("In Topbase value is : ",self.i)
        print("In derivedOne value is : ",self.j)
```

```
oderTwo=derivedTwo(10,20,30)
oderTwo.Display()
print("**********************************************************")
derivedOne.Display(oderTwo)
Topbase.Display(oderTwo)
print("**********************************************************")
```

Multiple Inheritance

- Python supports a limited form of multiple inheritance as well

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
     . . .
    <statement-N>
```

- For old-style classes, the only rule is depth-first, left-to-right.
- Thus, if an attribute is not found in DerivedClassName, it is searched in Base1,
- then (recursively) in the base classes of Base1, and only if it is not found there, it is searched in Base2, and so on.

Multiple Inheritance: Example

```
class   base1:
    def __init__(self, name):
        self.Name=name
        print("In Base1")
    def __del__(self):
        print("From Base1 Bye...")

class   base2:
    def __init__(self,phone):
        self.Phone=phone
        print("In Base2")

    def __del__(self):
        print("From Base2 Bye...")
```

Multiple Inheritance: Example

```
class   derived(base1,base2):
    def __init__(self, name,phone,address):
        self.Add=address
        base1.__init__(self,name)
        base2.__init__(self,phone)
        print("In Derived")
          def __del__(self):
        print("From derived Bye...")

    def Display(self):
        print(self.Name)
        print(self.Phone)
        print(self.Add)

oDer=derived("Vikas",124356,"Bangalore")
oDer.Display()
```

Method Overriding

```
#!/usr/bin/python

class Parent:        # define parent class
   def myMethod(self):
      print 'Calling parent method'

class Child(Parent): # define child class
   def myMethod(self):
      print 'Calling child method'

c = Child()          # instance of child
Parent.myMethod(c)        # child calls overridden method
Child.myMethod(c)
```

Special Functions in Python

- Class functions that begins with double underscore (__) are called special functions in Python.

- The __init__() function
- The __str__() function

Operator	Expression	Internally
Addition	p1 + p2	p1.__add__(p2)
Subtraction	p1 - p2	p1.__sub__(p2)
Multiplication	p1 * p2	p1.__mul__(p2)
Power	p1 ** p2	p1.__pow__(p2)

Operator overloading in python

Operator	Expression	Internally
Division	p1 / p2	p1.__truediv__(p2) or p1.__div__(p2)
Floor Division	p1 // p2	p1.__floordiv__(p2)
Remainder (modulo)	p1 % p2	p1.__mod__(p2)
Bitwise Left Shift	p1 << p2	p1.__lshift__(p2)
Bitwise Right Shift	p1 >> p2	p1.__rshift__(p2)
Bitwise AND	p1 & p2	p1.__and__(p2)
Bitwise OR	p1 | p2	p1.__or__(p2)
Bitwise XOR	p1 ^ p2	p1.__xor__(p2)
Bitwise NOT	~p1	p1.__invert__()

Overloading the + Operator

```
class Point:
    # previous definitions...

    def __add__(self,other):
        x = self.x + other.x
        y = self.y + other.y
        return Point(x,y)
```

```
class Point:
        # previous definitions...
    def __lt__(self,other):
        self_mag = (self.x ** 2) + (self.y ** 2)
        other_mag = (other.x ** 2) + (other.y ** 2)
        return self_mag < other_mag
```

Debugger -- pdb

Agenda

- How does the debugger work?
- Debugger Commands
- Setting (conditional) breakpoints
- Single stepping at the source line level,
- Inspection of stack frames,
- Source code listing

Debugger -- pdb

- The module pdb defines an interactive source code debugger for Python programs

- . Typical usage to run a program under control of the debugger is:

```
>>> import pdb
>>> import mymodule
>>> pdb.run('mymodule.test()')
```

- pdb.py can also be invoked as a script to debug other scripts. For example:

- python3 -m pdb myscript.py

Debugger -- pdb

- Command tp print the documentation on pdb
- `>>> import pdb`
- `>>> pdb.help()`

- Command to print the short documentation on pdb commands

- `(Pdb) help clear`
- `cl(ear) filename:lineno`
- `cl(ear) [bpnumber [bpnumber...]]`

- Debugger commands
- =================
- h(elp)
-     Without argument, print the list of available commands.
-

Debugger -- pdb

- w(here)

  Print a stack trace, with the most recent frame at the bottom. An arrow indicates the "current frame", which determines the context of most commands. 'bt' is an alias for this command.

- d(own) [count]

  Move the current frame count (default one) levels down in the stack trace (to a newer frame).

- u(p) [count]

  Move the current frame count (default one) levels up in the stack trace (to an older frame).

Debugger -- pdb

- b(reak) [([filename:]lineno | function) [, condition]]

  Without argument, list all breaks.

cl(ear) filename:lineno

cl(ear) [bpnumber [bpnumber...]]

  With a space separated list of breakpoint numbers, clear those breakpoints. Without argument, clear all breaks

- disable bpnumber [bpnumber ...]

  Disables the breakpoints given as a space separated list of breakpoint numbers it remains in the list of breakpoints and can be (re-)enabled.

Debugger -- pdb

- enable bpnumber [bpnumber ...]

  Enables the breakpoints given as a space separated list of breakpoint numbers

- s(tep)

  Execute the current line, stop at the first possible occasion (either in a function that is called or in the current function).

- n(ext)

  Continue execution until the next line in the current function is reached or it returns.

j(ump) lineno

Set the next line that will be executed. Only available in the bottom-most frame.

Debugger -- pdb

- l(ist) [first [,last] | .]

  List source code for the current file. Without arguments, list 11 lines around the current line or continue the previous listing.

- display [expression]

  Display the value of the expression if it changed, each time execution stops in the current frame.

Example 1:

```
import pdb;

def add( a,b) :
    return a+b

def sub(a,b):
    return a-b

def mul(a,b):
    return a*b

def div(a,b):
    return a/b
```

```
pdb.set_trace()

val1=add(5,10)
val2=sub(5,10)
val3=div(5,10)
val4=mul(5,10)


print(val1)
print(val2)
print(val3)
print(val4)
```

Example 2:

```
import pdb

def fun(a):
    ret1=foo(a*10)
    print("Return from foo",ret1)
    print("a in  fun is :",a)
    return ret1

def foo(a):
    ret2=bar(a*10)
    print("Return from bar",ret2)
    print("a in  bar is :",a)
    return ret2

def bar(a):
    ret3=(a*10)
    print("value in bar is ",ret3)
    print("a in  fun is :",a)
    return ret3

pdb.set_trace()
fun(10)
```

Thank you

TATA ELXSI

ITPB Road Whitefield

Bangalore 560 048 India
Tel +91 80 2297 9123
Fax +91 80 2841 1474
e-mail info@tataelxsi.com

www.tataelxsi.com