An introduction to the Python programming language

Prabhu Ramachandran

Department of Aerospace Engineering
IIT Bombay

January 16, 2007

Outline

Outline

Introduction

- Creator and BDFL: Guido van Rossum
- BDFL == Benevolent Dictator For Life
- Conceived in December 1989
- The name “Python”: Monty Python’s Flying Circus
- Current stable version of Python is 2.5.x
- PSF license (like BSD: no strings attached)
- Highly cross platform
- Runs on the Nokia series 60!

Resources

- Available as part of any sane GNU/Linux distribution
- Web: `http://www.python.org`
- Documentation: `http://www.python.org/doc`
- Free Tutorials:
  - Official Python tutorial: `http://docs.python.org/tut/tut.html`
  - Byte of Python: `http://www.byteofpython.info/`
  - Dive into Python: `http://diveintopython.org/`

Why Python?

- High level, interpreted, modular, OO
- Easy to learn
- Easy to read code
- Much faster development cycle
- Powerful interactive interpreter
- Rapid application development
- Powerful standard library
- Interfaces well to C++, C and FORTRAN libraries
- In short: there is little you can't do with it

A quote

I came across Python and its Numerical extension in 1998 . . . I quickly fell in love with Python programming which is a remarkable statement to make about a programming language. If I had not seen others with the same view, I might have seriously doubted my sanity.

– Travis Oliphant (creator of NumPy)

Why not ***lab?

- Open Source, Free
- Portable
- Python is a real programming language: large and small programs
- Can do much more than just array and math
  - Wrap large C++ codes
  - Build large code bases via SCons
  - Interactive data analysis/plotting
  - Parallel application
  - Job scheduling on a custom cluster
  - Miscellaneous scripts

Why not Python?

- Can be slow for high-performance applications
- This can be fairly easily overcome by using C/C++/FORTRAN extensions

Use cases

- NASA: Python Streamlines Space Shuttle Mission Design
- AstraZeneca Uses Python for Collaborative Drug Discovery
- ForecastWatch.com Uses Python To Help Meteorologists
- Industrial Light & Magic Runs on Python
- Zope: Commercial grade CMS
- RedHat: install scripts, sys-admin tools
- Numerous success stories:
  `http://www.pythonology.com/success`

Before we begin

- This is only an introduction
- Python is a full-fledged programming language
- Please read the tutorial
- It is very well written and can be read in one afternoon

Outline

Preliminaries: The Interpreter

- Python is interpreted
- Interpreted vs. Compiled
- Interpreted languages allow for rapid testing/prototyping
- Dynamically typed
- Full introspection of code at runtime
- Did I say dynamic?
- Does not force OO or a particular programming paradigm down your throat!

Preliminaries: IPython

- Recommended interpreter, IPython: `http://ipython.scipy.org`
- Better than the default Python shell
- Supports tab completion by default
- Easier object introspection
- Shell access!
- Command system to allow extending its own behavior
- Supports history (across sessions) and logging
- Can be embedded in your own Python code
- Support for macros
- A flexible framework for your own custom interpreter
- Other miscellaneous conveniences
- We'll get back to this later

Preliminaries . . .

- Will follow the Official Python tutorial
- No lexical blocking
- Indentation specifies scope
- Leads to easier to read code!

Using the interpreter

- Starting up: `python` or `ipython`
- Quitting: `Control-D` or `Control-Z` (on Win32)
- Can use it like a calculator
- Can execute one-liners via the `-c` option: `python -c "print 'hello world'"`
- Other options via `python -h`

Basic concepts

- Dynamically typed
- Assignments need not specify a type

```
a = 1
a = 1.1
a = "foo"
a = SomeClass()
```

- Comments:

```
a = 1  # In-line comments
# Comment in a line to itself.
a = "# This is not a comment!"
```

Basic concepts

- No lexical scoping
- Scope determined by indentation

```
for i in range(10):
    print "inside loop:",
    # Do something in the loop.
    print i, i*i
print "loop is done!" # This is outside the loop
```

- Assignment to an object is by reference
- Essentially, names are bound to objects

Outline

Basic types

- Basic objects: numbers (float, int, long, complex), strings, tuples, lists, dictionaries, functions, classes, types etc.
- Types are of two kinds: mutable and immutable
- Immutable types: numbers, strings, `None` and tuples
- Immutables cannot be changed “in-place”
- Mutable types: lists, dictionaries, instances, etc.
- Mutable objects can be “changed”

What is an object?

- A loose and informal but handy description
- An object is a particular instance of a general class of things

Real world example

- Consider the class of cars made by Honda
- A Honda Accord on the road, is a particular instance of the general class of cars
- It is an object in the general sense of the term

Objects in a programming language

- The object in the computer follows a similar idea
- An object has attributes and behavior
- Object contains or manages data (attributes) and has methods (behavior)
- Together this lets one create representation of “real” things on the computer
- Programmers then create objects and manipulate them through their methods to get things done
- In Python everything is essentially an object – you don’t have to worry about it

Numbers

```
>>> a = 1 # Int.
>>> l = 1000000L # Long
>>> e = 1.01325e5 # float
>>> f = 3.14159 # float
>>> c = 1+1j # Complex!
>>> print f*c/a
(3.14159+3.14159j)
>>> print c.real, c.imag
1.0 1.0
>>> abs(c)
1.4142135623730951
```

Boolean

```
>>> t = True
>>> f = not t
False
>>> f or t
True
>>> f and t
False
```

Strings

```
s = 'this is a string'
s = 'This one has "quotes" inside!'
s = "The reverse with 'single-quotes' inside!"
l = "A long string spanning several lines\
one more line\
yet another"
t = """A triple quoted string
does not need to be escaped at the end and
"can have nested quotes" and whatnot."""
```

Strings

```
>>> word = "hello"
>>> 2*word + " world" # Arithmetic on strings
hellohello world
>>> print word[0] + word[2] + word[-1]
hlo
>>> # Strings are "immutable"
... word[0] = 'H'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
>>> len(word) # The length of the string
5
>>> s = u'Unicode strings!'  # unicode string
```

More on strings

```
>>> a = 'hello world'
>>> a.startswith('hell')
True
>>> a.endswith('ld')
True
>>> a.upper()
'HELLO WORLD'
>>> a.upper().lower()
'hello world'
>>> a.split()
['hello', 'world']
>>> ''.join(['a', 'b', 'c'])
'abc'
>>> x, y = 1, 1.234
>>> 'x is %s, y is %s'%(x, y)
'x is 1, y is 1.234'
>>> # could also do: 'x is %d, y is %f'%(x, y)
```

See `http://docs.python.org/lib/typesseq-strings.html`

Lists

- Lists are mutable
- Items are indexed at 0
- Last element is -1
- Length: `len(list)`

List: examples

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> a[0]
'spam'
>>> a[3]
1234
>>> a[-2]
100
>>> a[1:-1]
['eggs', 100]
>>> a[:2] + ['bacon', 2*2]
['spam', 'eggs', 'bacon', 4]
>>> 2*a[:3] + ['Boe!']
['spam', 'eggs', 100, 'spam', 'eggs', 100, 'Boe!']
```

Lists are mutable

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> a[2] = a[2] + 23
>>> a
['spam', 'eggs', 123, 1234]
>>> a = ['spam', 'eggs', 100, 1234]
>>> a[0:2] = [1, 12] # Replace some items
>>> a
[1, 12, 123, 1234]
>>> a[0:2] = [] # Remove some items
>>> a
[123, 1234]
```

List methods

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> len(a)
4
>>> a.reverse()
>>> a
[1234, 100, 'eggs', 'spam']
>>> a.append(['x', 1]) # Lists can contain lists.
>>> a
[1234, 100, 'eggs', 'spam', ['x', 1]]
>>> a.extend([1,2]) # Extend the list.
>>> a
[1234, 100, 'eggs', 'spam', ['x', 1], 1, 2]
```

Tuples

```
>>> t = (0, 1, 2)
>>> print t[0], t[1], t[2], t[-1], t[-2], t[-3]
0 1 2 2 1 0
>>> # Tuples are immutable!
... t[0] = 1
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
```

Dictionaries

- Associative arrays/mappings
- Indexed by “keys” (keys must be immutable)
- `dict[key] = value`
- `keys()` returns all keys of the dict
- `values()` returns the values of the dict
- `has_key(key)` returns if `key` is in the dict

Dictionaries: example

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> tel.keys()
['guido', 'irv', 'jack']
>>> tel.has_key('guido')
True
```

Outline

Control flow

- Control flow is primarily achieved using the following:
- `if/elif/else`
- `for`
- `while`
- `break, continue, else` may be used to further control
- `pass` can be used when syntactically necessary but nothing needs to be done

If example

```
x = int(raw_input("Please enter an integer: "))
# raw_input asks the user for input.
# int() typecasts the resulting string into an int.
if x < 0:
     x = 0
     print 'Negative changed to zero'
elif x == 0:
     print 'Zero'
elif x == 1:
     print 'Single'
else:
     print 'More'
```

`If` example

```
>>> a = ['cat', 'window', 'defenestrate']
>>> if 'cat' in a:
...     print "meaw"
...
meaw
>>> pets = {'cat': 1, 'dog':2, 'croc': 10}
>>> if 'croc' in pets:
...     print pets['croc']
...
10
```

`for` example

```
>>> a = ['cat', 'window', 'defenestrate']
>>> for x in a:
...     print x, len(x)
...
cat 3
window 6
defenestrate 12
>>> knights = {'gallahad': 'the pure',
... 'robin': 'the brave'}
>>> for k, v in knights.iteritems():
...     print k, v
...
gallahad the pure
robin the brave
```

for and range example

```
>>> for i in range(5):
...     print i, i*i
...
0 0
1 1
2 4
3 9
4 16
>>> a = ['a', 'b', 'c']
>>> for i, x in enumerate(a):
...     print i, x
...
0 'a'
1 'b'
2 'c'
```

`while` example

```
>>> # Fibonacci series:
... # the sum of two elements defines the next
... a, b = 0, 1
>>> while b < 10:
...         print b
...         a, b = b, a+b
...
1
1
2
3
5
8
```

Functions

- Support default and keyword arguments
- Scope of variables in the function is local
- Mutable items are passed by reference
- First line after definition may be a documentation string (recommended!)
- Function definition and execution defines a name bound to the function
- You *can* assign a variable to a function!

Functions: examples

```
>>> def fib(n): # write Fibonacci series up to n
...     """Print a Fibonacci series up to n."""
...     a, b = 0, 1
...     while b < n:
...         print b,
...         a, b = b, a+b
...
>>> # Now call the function we just defined:
... fib(2000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
>>> f = fib  # Assign a variable to a function
>>> f(10)
1 1 2 3 5 8
```

Functions: default arguments

```
def ask_ok(prompt, retries=4, complaint='Yes or no!'):
    while True:
        ok = raw_input(prompt)
        if ok in ('y', 'ye', 'yes'):
            return True
        if ok in ('n', 'no', 'nop', 'nope'):
            return False
        retries = retries - 1
        if retries < 0:
            raise IOError, 'bad user'
        print complaint
```

Functions: keyword arguments

```python
def parrot(voltage, state='a stiff',
           action='voom', type='Norwegian Blue'):
    print "-- This parrot wouldn't", action,
    print "if you put", voltage, "Volts through it."
    print "-- Lovely plumage, the", type
    print "-- It's", state, "!"

parrot(1000)
parrot(action = 'VOOOOOM', voltage = 1000000)
parrot('a thousand', state = 'pushing up the daisies')
parrot('a million', 'bereft of life', 'jump')
```

Outline

Modules

- Define variables, functions and classes in a file with a `.py` extension
- This file becomes a module!
- Modules are searched in the following:
  - Current directory
  - Standard: `/usr/lib/python2.3/site-packages/` etc.
  - Directories specified in PYTHONPATH
  - `sys.path`: current path settings (from the `sys` module)
- The `import` keyword "loads" a module
- One can also use:
  `from module import name1, name2, name2`
  where `name1` etc. are names in the module, "module"
- `from module import *` — imports everything from module, use only in interactive mode

Modules: example

```
# —— foo.py ——
some_var = 1
def fib(n): # write Fibonacci series up to n
    """Print a Fibonacci series up to n."""
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a+b
# EOF

>>> import foo
>>> foo.fib(10)
1 1 2 3 5 8
>>> foo.some_var
1
```

Namespaces

- A mapping from names to objects
- Modules introduce a namespace
- So do classes
- The running script's namespace is `__main__`
- A modules namespace is identified by its name
- The standard functions (like `len`) are in the `__builtin__` namespace
- Namespaces help organize different names and their bindings to different objects

Exceptions

- Python's way of notifying you of errors
- Several standard exceptions: `SyntaxError`, `IOError` etc.
- Users can also `raise` errors
- Users can create their own exceptions
- Exceptions can be "caught" via `try/except` blocks

Exception: examples

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: integer 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: cannot concatenate 'str' and 'int' objects
```

Exception: examples

```
>>> while True:
...     try:
...         x = int(raw_input("Enter a number: "))
...         break
...     except ValueError:
...         print "Invalid number, try again..."
...
>>> # To raise exceptions
... raise ValueError, "your error message"
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
ValueError: your error message
```

Classes

- Class definitions when executed create class objects
- Classes have attributes and may be instantiated
- Instantiating the class object creates an instance of the class
- All attributes are accessed via the `value.attribute` syntax
- Both class and instance attributes are supported
- *Methods* represent the behavior of an object: crudely think of them as functions “belonging” to the object
- All methods in Python are “virtual”
- Classes may be subclassed (heh!)
- Multiple inheritance is supported
- There are no special public and private attributes

Classes: examples

```
class MyClass(object):
    "Example class (this is the class docstring)"
    i = 12345 # A class attribute
    def f(self):
        "This is the method docstring"
        return 'hello world'

>>> a = MyClass() # creates an instance
>>> a.f()
'hello world'
>>> # a.f() is equivalent to MyClass.f(a)
... # This also explains why f has a 'self' argument.
... MyClass.f(a)
'hello world'
```

Classes (continued)

- `self` is conventionally the first argument for a method
- In previous example, `a.f` is a method object
- When `a.f` is called, it is passed the instance `a` as the first argument
- If a method called `__init__` exists, it is called when the object is created
- If a method called `__del__` exists, it is called before the object is garbage collected
- Instance attributes are set by simply "setting" them in `self`
- Other special methods (by convention) like `__add__` let you define numeric types: `http://docs.python.org/ref/numeric-types.html`

Classes: examples

```
class Bag(MyClass): # Shows how to derive classes
    def __init__(self): # called on object creation.
        self.data = [] # an instance attribute
    def add(self, x):
        self.data.append(x)
    def addtwice(self, x):
        self.add(x)
        self.add(x)
>>> a = Bag()
>>> a.f() # Inherited method
'hello world'
>>> a.add(1)
>>> a.addtwice(2)
>>> a.data
[1, 2, 2]
```

Stand-alone scripts

Consider a file `f.py`:

```
#!/usr/bin/env python
"""Module level documentation."""
# First line tells the shell that it should use Python
# to interpret the code in the file.
def f():
    print "f"

# Check if we are running standalone or as module.
# When imported, __name__ will not be '__main__'
if __name__ == '__main__':
    # This is not executed when f.py is imported.
    f()
```

Outline

File handling

```
>>> # Reading files:
... f = open('/path/to/file_name')
>>> data = f.read() # Read entire file.
>>> line = f.readline() # Read one line.
>>> # Read entire file appending each line into a list
... lines = f.readlines()
>>> f.close() # close the file.
>>> # Writing files:
... f = open('/path/to/file_name', 'w')
>>> f.write('hello world\n')
```

- `tell()`: returns int of current position
- `seek(pos)`: moves current position to specified byte
- Call `close()` when done using a file

Math

- `math` module provides basic math routines for floats
- `cmath` module provides math routies for complex numbers
- `random`: provides pseudo-random number generators for various distributions
- These are always available and part of the standard library
- More serious math is provided by the NumPy/SciPy modules – these are not standard and need to be installed separately

Timing and profiling

- Timing code: use the `time` module
- Read up on `time.time()` and `time.clock()`
- `timeit`: is a better way of doing timing
- IPython has handy `time` and `timeit` macros (type `timeit?` for help)
- IPython lets you debug and profile code via the `run` macro (type `run?` on the prompt to learn more)

Odds and ends

- `dir([object])` function: attributes of given object
- `type(object)`: returns type information
- `str(), repr()`: convert object to string representation
- `isinstance, issubclass`
- `assert` statements let you do debugging assertions in code
- `csv` module: reading and writing CSV files
- `pickle`: lets you save and load Python objects (serialization)
- `os.path`: common path manipulations
- Check out the Python Library reference: `http://docs.python.org/lib/lib.html`

Outline

The `numpy` module

- Manipulating large Python lists for scientific computing is slow
- Most complex computations can be reduced to a few standard operations
- The `numpy` module provides:
  - An efficient and powerful array type for various common data types
  - Abstracts out the most commonly used standard operations on arrays
- Numeric was the first, then came `numarray`. `numpy` is the latest and is the future
- This course uses `numpy` and only covers the absolute basics

Basic concepts

- `numpy` arrays are of a fixed size (`arr.size`) and have the same type (`arr.dtype`)
- `numpy` arrays may have arbitrary dimensionality
- The `shape` of an array is the extent (length) of the array along each dimension
- The `rank(arr)` of an array is the “dimensionality” of the array
- The `arr.itemsize` is the number of bytes (8-bits) used for each element of the array
- Note: The `shape` and `rank` may change as long as the `size` of the array is fixed
- Note: `len(arr) != arr.size` in general
- Note: By default array operations are performed elementwise
- Indices start from 0

Examples of numpy

```
# Simple array math example
>>> from numpy import *
>>> a = array([1,2,3,4])
>>> b = array([2,3,4,5])
>>> a + b # Element wise addition!
array([3, 5, 7, 9])

>>> print pi, e  # Pi and e are defined.
3.14159265359 2.71828182846
# Create array from 0 to 10
>>> x = arange(0.0, 10.0, 0.05)
>>> x *= 2*pi/10 # multiply array by scalar value
array([ 0.,0.0314,...,6.252])
# apply functions to array.
>>> y = sin(x)
```

More examples of numpy

```
# Size, shape, rank, type etc.
>>> x = array([1., 2, 3, 4])
>>> size(x)
4
>>> x.dtype # or x.dtype.char
'd'
>>> x.shape
(4,)
>>> print rank(x), x.itemsize
1 8
>>> x.tolist()
[1.0, 2.0, 3.0, 4.0]
# Array indexing
>>> x[0] = 10
>>> print x[0], x[-1]
10.0 4.0
```

Multi-dimensional arrays

```
>>> a = array([[ 0, 1, 2, 3],
...            [10,11,12,13]])
>>> a.shape # (rows, columns)
(2, 4)
# Accessing and setting values
>>> a[1,3]
13
>>> a[1,3] = -1
>>> a[1] # The second row
array([10,11,12,-1])

# Flatten/ravel arrays to 1D arrays
>>> a.flat  # or ravel(a)
array([0,1,2,3,10,11,12,-1])
# Note: flat references original memory
```

Slicing arrays

```
>>> a = array([[1,2,3], [4,5,6], [7,8,9]])
>>> a[0,1:3]
array([2, 3])
>>> a[1:,1:]
array([[5, 6],
       [8, 9]])
>>> a[:,2]
array([3, 6, 9])
# Striding...
>>> a[0::2,0::2]
array([[1, 3],
       [7, 9]])
# All these slices are references to the same memory!
```

Array creation functions

- `array(object, dtype=None, copy=1,order=None, subok=0,ndmin=0)`
- `arange(start, stop=None, step=1, dtype=None)`
- `linspace(start, stop, num=50, endpoint=True, retstep=False)`
- `ones(shape, dtype=None, order='C')`
- `zeros((d1,...,dn),dtype=float,order='C')`
- `identity(n)`
- `empty((d1,...,dn),dtype=float,order='C')`
- `ones_like(x), zeros_like(x), empty_like(x)`

Array math

- Basic elementwise math (given two arrays `a, b`):
  - `a + b` → `add(a, b)`
  - `a - b,` → `subtract(a, b)`
  - `a * b,` → `multiply(a, b)`
  - `a / b,` → `divide(a, b)`
  - `a % b,` → `remainder(a, b)`
  - `a ** b,` → `power(a, b)`
- Inplace operators: `a += b`, or `add(a, b, a)` etc.
- Logical operations: `equal (==)`, `not_equal (!=)`, `less (<)`, `greater (>)` etc.
- Trig and other functions: `sin(x), arcsin(x), sinh(x), exp(x), sqrt(x)` etc.
- `sum(x, axis=0), product(x, axis=0)`: sum and product of array elements
- `dot(a, b)`

Advanced

- Only scratched the surface of `numpy`
- Ufunc methods: `reduce, accumulate, outer, reduceat`
- Typecasting
- More functions: `take, choose, where, compress, concatenate`
- Array broadcasting and `None`

Outline

About `matplotlib`

- Easy to use, scriptable, "Matlab-like" 2D plotting
- Publication quality figures and interactive capabilities
- Plots, histograms, power spectra, bar charts, errorcharts, scatterplots, etc.
- Also does polar plots, maps, contours
- Support for simple TeX markup
- Multiple output backends (images, EPS, SVG, wx, Agg, Tk, GTK)
- Cross-platform: Linux, Win32, Mac OS X
- Good idea to use via IPython: `ipython -pylab`
- From scripts use: `import pylab`

Basic plotting with `matplotlib`

```
>>> x = arange(0, 2*pi, 0.05)
>>> plot(x, sin(x)) # Same as plot(x, sin(x), 'b-')
>>> plot(x, sin(x), 'ro')
>>> axis([0,2*pi, -1,1])
>>> xlabel(r'$\chi$', color='g')
>>> ylabel(r'sin($\chi$)', color='r')
>>> title('A simple figure', fontsize=20)
>>> savefig('/tmp/test.eps')
# Multiple plots in one figure
>>> t = arange(0.0, 5.2, 0.2)
# red dashes, blue squares and green triangles
>>> plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^')
```

More on plotting

```
# Set properties of objects:
>>> plot(x, sin(x), linewidth=2.0, color='r')
>>> l, = plot(x, sin(x))
>>> setp(l, linewidth=2.0, color='r')
>>> l.set_linewidth(2.0); l.set_color('r')
>>> draw() # Redraws current figure.
>>> setp(l) # Prints available properties
>>> close() # Closes the figure.
# Multiple figures:
>>> figure(1); plot(x, sin(x))
>>> figure(2); plot(x, tanh(x))
>>> figure(1); title('Easy as 1,2,3')
```

More on plotting ...

```
>>> figure(1)
>>> subplot(211) # Same as subplot(2, 1, 1)
>>> plot(x, cos(5*x)*exp(-x))
>>> subplot(2, 1, 2)
>>> plot(x, cos(5*x), 'r--', label='cosine')
>>> plot(x, sin(5*x), 'g--', label='sine')
>>> legend() # Or legend(['cosine', 'sine'])
>>> text(1,0, '(1,0)')
>>> axes = gca() # Current axis
>>> fig = gcf() # Current figure
```

More information

- More information here: `http://matplotlib.sf.net`
- `http://matplotlib.sf.net/tutorial.html`
- `http://matplotlib.sf.net/screenshots.html`

Outline

Using `SciPy`

- SciPy is Open Source software for mathematics, science, and engineering
- `import scipy`
- Built on NumPy/Numeric
- Provides modules for statistics, optimization, integration, linear algebra, Fourier transforms, signal and image processing, genetic algorithms, ODE solvers, special functions, and more
- Used widely by scientists world over
- Details are beyond the scope of this tutorial

Outline

Standard library

- Very powerful
- “Batteries included”
- Example standard modules taken from the tutorial
  - Operating system interface: `os`
  - System, Command line arguments: `sys`
  - Regular expressions: `re`
  - Math: `math`, `random`
  - Internet access: `urllib2`, `smtplib`
  - Data compression: `zlib`, `gzip`, `bz2`, `zipfile`, and `tarfile`
  - Unit testing: `doctest` and `unittest`
  - And a whole lot more!
- Check out the Python Library reference: `http://docs.python.org/lib/lib.html`

Stdlib: examples

```
>>> import os
>>> os.system('date')
Fri Jun 10 22:13:09 IST 2005
0
>>> os.getcwd()
'/home/prabhu'
>>> os.chdir('/tmp')
>>> import os
>>> dir(os)
<returns a list of all module functions>
>>> help(os)
<extensive manual page from module's docstrings>
```

Stdlib: examples

```
>>> import sys
>>> # Print the list of command line args to Python
... print sys.argv
['']
>>> import re # Regular expressions
>>> re.findall(r'\bf[a-z]*',
... 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
>>> re.sub(r'(\b[a-z]+) \1', r'\1',
... 'cat in the the hat')
'cat in the hat'
```

Stdlib: examples

```
>>> import math
>>> math.cos(math.pi / 4.0)
0.70710678118654757
>>> math.log(1024, 2)
10.0
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'pear'
```

Stdlib: examples

```
>>> import urllib2
>>> f = urllib2.urlopen('http://www.python.org/')
>>> print f.read(100)
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitio
<?xml-stylesheet href="./css/ht2html
```

Stdlib: examples

```
>>> import zlib
>>> s = 'witch which has which witches wrist watch'
>>> len(s)
41
>>> t = zlib.compress(s)
>>> len(t)
37
>>> zlib.decompress(t)
'witch which has which witches wrist watch'
>>> zlib.crc32(t)
-1438085031
```

Summary

- Introduced Python
- Basic syntax
- Basic types and data structures
- Control flow
- Functions
- Modules
- Exceptions
- Classes
- Standard library