



python :  
powered



python



# python

- Simple
  - Python is a simple and minimalistic language in nature
  - Reading a **good** python program should be like reading English
  - Its Pseudo-code nature allows one to concentrate on the problem rather than the language
- Easy to Learn
- Free & Open source
  - Freely distributed and Open source
  - Maintained by the Python community  
<http://www.python.org/community/>
- High Level Language – memory management
- Portable – \*runs on anything c code will



# python

- Interpreted
  - You run the program straight from the source code.
  - Python program → Bytecode → a platform's native language
  - You can just copy over your code to another system and it will automatically work with python platform
- Object-Oriented
  - Simple and additionally supports procedural programming
- Extensible – easily import other code
- Embeddable – easily place your code in non-python programs
- Extensive libraries
  - (i.e. reg. expressions, doc generation, CGI, ftp, web browsers, ZIP, WAV, cryptography, etc...) (wxPython, Twisted, Python Imaging library)



# python Timeline/History



- Python was conceived in the late 1980s.
  - Guido van Rossum, Benevolent Dictator For Life (仁慈独裁者)
  - Rossum is Dutch, born in Netherlands
  - Descendant of ABC, he wrote glob() func in UNIX
  - M.D. @ U of Amsterdam, worked for CWI, NIST, CNRI, Google
  - Also, helped develop the ABC programming language
- In 1991 python 0.9.0 was published and reached the masses through alt.sources
  - The [alt.sources](#) newsgroup is intended to be a repository for source-code of all sorts that people wish to distribute and share with other people.
- In January of 1994 python 1.0 was released
  - Functional programming tools like lambda, map, filter, and reduce
  - comp.lang.python formed, greatly increasing python's userbase



# python Timeline/History

- In 1995, python 1.2 was released.
- By version 1.4 python had several new features
  - Keyword arguments (similar to those of common lisp)
  - Built-in support for complex numbers
  - Basic form of data-hiding through name mangling (easily bypassed)
    - private, protected, public
- Computer Programming for Everybody initiative
  - Make programming accessible to more people, with basic “literacy” similar to those required for English and math skills for some jobs.
  - Project was funded by DARPA (Defense Advanced Research Projects Agency)



# python Timeline/History

- In 2000, Python 2.0 was released.
  - Introduced list comprehensions similar to Haskell
    - Haskell is a modern functional language (like lisp)
  - Introduced garbage collection
- In 2001, Python 2.2 was released.
  - Included unification of types and classes into one hierarchy, making python's object model purely Object-oriented
  - Generators were added (function-like iterator behavior)
    - **iterator** is an object that enables a programmer to traverse a container.
- Standards

# Running Python

- There are **three** different ways to start Python:

## (1) Interactive Interpreter:

- You can enter **python** and start coding right away in the interactive interpreter by starting it from the command line.

```
$python          # Unix/Linux  
  
or  
  
python$        # Unix/Linux  
  
or  
  
C:>python      # Windows/DOS
```

# Interactive Interpreter

- Here is the list of all the available command line options:

| Option | Description  |
|--------|--|
| -d     | provide debug output   |
| -O     | generate optimized bytecode (resulting in .pyo files)  |
| -S     | do not run import site to look for Python paths on startup                                     |
| -v     | verbose output (detailed trace on import statements)   |
| -X     | disable class-based built-in exceptions (just use strings); obsolete starting with version 1.6 |
| -c cmd | run Python script sent in as cmd string  |
| file   | run Python script from given file  |



# Script from the Command-line

- A Python script can be executed at **command line** by invoking the interpreter on your application, as in the following:

```
$python script.py          # Unix/Linux
```

```
or
```

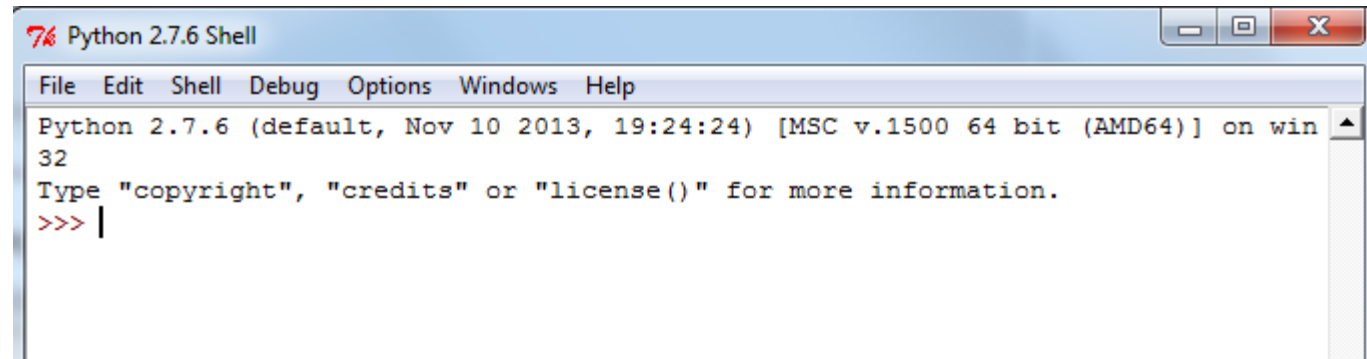
```
python% script.py         # Unix/Linux
```

```
or
```

```
C:>python script.py      # Windows/DOS
```

# Integrated Development Environment

- You can run Python from a graphical user interface (GUI) environment as well.
  - All you need is a GUI application on your system that supports Python.
- **Unix:** IDLE is the very first Unix IDE for Python.
- **Windows:** PythonWin is the first Windows interface for Python and is an IDE with a GUI.
- **Macintosh:** The Macintosh version of Python along with the IDLE IDE is available from the main website, downloadable as either MacBinary or BinHex'd files.



```
Python 2.7.6 Shell
File Edit Shell Debug Options Windows Help
Python 2.7.6 (default, Nov 10 2013, 19:24:24) [MSC v.1500 64 bit (AMD64)] on win
32
Type "copyright", "credits" or "license()" for more information.
>>> |
```

# Python Identifiers

- A Python identifier is a name used to identify a variable, function, class, module or other object.
- An identifier starts with a letter **A to Z** or **a to z** or an **underscore ( \_ )** followed by **zero or more letters, underscores and digits (0 to 9)**.
- Python does not allow punctuation characters such as **@**, **\$** and **%** within identifiers.
- Python is a **case sensitive** programming language.
  - Thus, **Manpower** and **manpower** are two different identifiers in Python.

# Python Identifiers

- Here are following identifier naming convention for Python:
  - Class names start with an uppercase letter and all other identifiers with a lowercase letter.
  - Starting an identifier with a **single leading underscore** (`_`) indicates by convention that the identifier is meant to be private.
    - `_single_leading_underscore`: weak "internal use" indicator.
  - Starting an identifier with **two leading underscores** (`__`) indicates a strongly private identifier.
    - a *double underscore* (`__`) is private; anything else isn't private.
  - If the identifier also ends with **two trailing underscores**, the identifier is a language-defined special name.

(e.g. `__spirit__` ).

# Reserved Words

|                       |                      |                     |
|-----------------------|----------------------|---------------------|
| <code>and</code>      | <code>exec</code>    | <code>not</code>    |
| <code>assert</code>   | <code>finally</code> | <code>or</code>     |
| <code>break</code>    | <code>for</code>     | <code>pass</code>   |
| <code>class</code>    | <code>from</code>    | <code>print</code>  |
| <code>continue</code> | <code>global</code>  | <code>raise</code>  |
| <code>def</code>      | <code>if</code>      | <code>return</code> |
| <code>del</code>      | <code>import</code>  | <code>try</code>    |
| <code>elif</code>     | <code>in</code>      | <code>while</code>  |
| <code>else</code>     | <code>is</code>      | <code>with</code>   |
| <code>except</code>   | <code>lambda</code>  | <code>yield</code>  |

# Lines and Indentation

- There are **no braces “( )”** to indicate blocks of code for class and function definitions or flow control.
- Blocks of code are denoted by **line indentation**, which is rigidly enforced.
- The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount.

```
if True:
    print "True"
else:
    print "False"
```

```
if True:
    print "Answer"
    print "True"
else:
    print "Answer"
    print "False"
```

# Multi-Line Statements

- Statements in Python typically end with a new line.
- Python does, however, allow the use of the **line continuation character (\)** to denote that the line should continue

```
total = item_one + \  
        item_two + \  
        item_three
```

## Quotation in Python

- Python accepts **single (')**, **double (")** and **triple (''' or ''')** quotes to denote string literals, as long as the same type of quote starts and ends the string.
- The triple quotes can be used to span the string across multiple lines

```
word = 'word'  
sentence = "This is a sentence."  
paragraph = """This is a paragraph. It is  
made up of multiple lines and sentences."""
```

# Comments in Python

- A **hash sign (#)** that is not inside a string literal begins a comment.
- All characters after the # and up to the physical line end are part of the comment and the Python interpreter ignores them.

```
#!/usr/bin/python

# First comment
print "Hello, Python!"; # second comment
```

## Multiple Statements on a Single Line

- The **semicolon (;)** allows multiple statements on the single line given that neither statement starts a new code block.

```
import sys; x = 'foo'; sys.stdout.write(x + '\n')
```



# Multiple Statement Groups as Suites

- A group of individual statements, which make a single code block are called **suites** in Python.
- Compound or complex statements, such as if, while, def, and class, are those which require a header line and a suite.
- Header lines begin the statement (with the keyword) and terminate with a **colon** ( : ) and are followed by one or more lines which make up the suite.

```
if expression :  
    suite  
elif expression :  
    suite  
else :  
    suite
```

# Command Line Arguments

- You may have seen, for instance, that many programs can be run so that they provide you with some basic information about how they should be run.
- Python enables you to do this with `-h`:

```
$ python -h
usage: python [option] ... [-c cmd | -m mod | file | -] [arg] ...
Options and arguments (and corresponding environment variables):
-c cmd : program passed in as string (terminates option list)
-d      : debug output from parser (also PYTHONDEBUG=x)
-E      : ignore environment variables (such as PYTHONPATH)
-h      : print this help message and exit

[ etc. ]
```

# Assigning Values to Variables

- Python variables do not have to be explicitly declared to reserve memory space.
- The declaration happens automatically when you assign a value to a variable.
  - The equal sign (=) is used to assign values to variables.
- The operand to the left of the = operator is the name of the variable and the operand to the right of the = operator is the value stored in the variable.

```
100
1000.0
John
```

```
#!/usr/bin/python

counter = 100           # An integer assignment
miles   = 1000.0       # A floating point
name    = "John"      # A string

print counter
print miles
print name
```



# Python types

- Int – 42- may be transparently expanded to long through 438324932L
- Float – 2.171892
- Complex –  $4 + 3j$
- Bool – True of False

# Multiple Assignment

- Python allows you to assign a single value to several variables simultaneously.

```
a = b = c = 1
```

```
a, b, c = 1, 2, "john"
```

# Standard Data Types:

- Python has five standard data types:
  1. Numbers (Number data types store numeric values.) 

```
var1 = 1  
var2 = 10
```
  2. String (Strings in Python are identified as a contiguous set of characters in between quotation marks(" ").)
  3. List (Lists are the most versatile of Python's compound data types.)
  4. Tuple (A tuple is another sequence data type that is similar to the list but it is immutable.)
  5. Dictionary (Python's dictionaries are kind of hash table type.)



# Python types

- Str, unicode – ‘MyString’, u‘MyString’
- List – [ 69, 6.9, ‘mystring’, True]
- Tuple – (69, 6.9, ‘mystring’, True) immutable
- Set/frozenset – set([69, 6.9, ‘str’, True])  
frozenset([69, 6.9, ‘str’, True]) immutable –no  
duplicates & unordered
- Dictionary or hash – {‘key 1’: 6.9, ‘key2’: False}  
- group of key and value pairs

# Python Strings

- Subsets of strings can be taken using the slice operator ( `[]` and `[:]` ) with indexes starting at 0 in the beginning of the string and working their way from -1 at the end.
- The plus ( `+` ) sign is the **string concatenation operator** and the asterisk ( `*` ) is the **repetition operator**.

```
#!/usr/bin/python
```

```
str = 'Hello World!'
```

```
print str          # Prints complete string
print str[0]       # Prints first character of the string
print str[2:5]     # Prints characters starting from 3rd to 5th
print str[2:]      # Prints string starting from 3rd character
print str * 2      # Prints string two times
print str + "TEST" # Prints concatenated string
```

```
Hello World!
```

```
H
```

```
llo
```

```
llo World!
```

```
Hello World!Hello World!
```

```
Hello World!TEST
```



# Python Lists

- A list contains items separated by commas (,) and enclosed within **square brackets** ([ ]).
- To some extent, lists are similar to arrays in C.
  - One difference between them is that all the items belonging to a list can be of different data type.
- The values stored in a list can be accessed using the **slice operator** ( [ ] and [ : ] ) with **indexes starting at 0 in the beginning** of the list and working their way **to end -1**.
- The plus ( + ) sign is the **list concatenation operator**, and the asterisk ( \* ) is the **repetition operator**.

# Python Lists

```
#!/usr/bin/python

list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
tinylist = [123, 'john']

print list          # Prints complete list
print list[0]       # Prints first element of the list
print list[1:3]     # Prints elements starting from 2nd till 3rd
print list[2:]      # Prints elements starting from 3rd element
print tinylist * 2  # Prints list two times
print list + tinylist # Prints concatenated lists
```

```
['abcd', 786, 2.23, 'john', 70.2000000000000003]
abcd
[786, 2.23]
[2.23, 'john', 70.2000000000000003]
[123, 'john', 123, 'john']
['abcd', 786, 2.23, 'john', 70.2000000000000003, 123, 'john']
```

# Python Tuples

- A tuple consists of a number of values separated by commas.
- Tuples are enclosed within parentheses ( ( ) ).
- The main differences between lists and tuples are:
  - Lists are enclosed in brackets ( [ ] ) and their elements and size can be changed, while tuples are enclosed in parentheses ( ( ) ) and cannot be updated.
  - Tuples can be thought of as **read-only** lists

```
#!/usr/bin/python
tuple = ( 'abcd', 786 , 2.23, 'john', 70.2 )
tinytuple = (123, 'john')

print tuple           # Prints complete list
print tuple[0]       # Prints first element of the list
print tuple[1:3]     # Prints elements starting from 2nd till 3rd
print tuple[2:]      # Prints elements starting from 3rd element
print tinytuple * 2  # Prints list two times
print tuple + tinytuple # Prints concatenated lists

('abcd', 786, 2.23, 'john', 70.20000000000000003)
abcd
(786, 2.23)
(2.23, 'john', 70.20000000000000003)
(123, 'john', 123, 'john')
('abcd', 786, 2.23, 'john', 70.20000000000000003, 123, 'john')
```

```
#!/usr/bin/python
tuple = ( 'abcd', 786 , 2.23, 'john', 70.2 )
list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
tuple[2] = 1000 # Invalid syntax with tuple
list[2] = 1000 # Valid syntax with list
```

# Tuple Example

```
>>> t = ([1, 2], [3, 4])
>>> t
([1, 2], [3, 4])
>>> t[0] = [10, 20]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

# Python Dictionary

- A dictionary key can be almost any Python type, but are usually numbers or strings.
  - Values, on the other hand, can be any arbitrary Python object.
- Dictionaries are enclosed by curly braces ( `{ }` ) and values can be assigned and accessed using square braces ( `[ ]` )

```
#!/usr/bin/python

dict = {}
dict['one'] = "This is one"
dict[2] = "This is two"

tinydict = {'name': 'john', 'code':6734, 'dept': 'sales'}

print dict['one']      # Prints value for 'one' key
print dict[2]         # Prints value for 2 key
print tinydict        # Prints complete dictionary
print tinydict.keys() # Prints all the keys
print tinydict.values() # Prints all the values
```

```
This is one
This is two
{'dept': 'sales', 'code': 6734, 'name': 'john'}
['dept', 'code', 'name']
['sales', 6734, 'john']
```

Python 2.7.6 Shell

File Edit Shell Debug Options Windows Help

Python 2.7.6 (default, Nov 10 2013, 19:24:24) [MSC v. 32

Type "copyright", "credits" or "license()" for more i

```
>>> dict = {'name': 'jojn', 'code':6734, 'dept': 'sal
```

```
>>> print dict
```

```
{'dept': 'sale', 'code': 6734, 'name': 'jojn'}
```

```
>>> |
```

| Function                           | Description  |
|------------------------------------|--|
| <code>int(x [,base])</code>        | Converts <code>x</code> to an integer. <code>base</code> specifies the base if <code>x</code> is a string.     |
| <code>long(x [,base] )</code>      | Converts <code>x</code> to a long integer. <code>base</code> specifies the base if <code>x</code> is a string. |
| <code>float(x)</code>              | Converts <code>x</code> to a floating-point number.  |
| <code>complex(real [,imag])</code> | Creates a complex number.  |
| <code>str(x)</code>                | Converts object <code>x</code> to a string representation.   |
| <code>repr(x)</code>               | Converts object <code>x</code> to an expression string.  |
| <code>eval(str)</code>             | Evaluates a string and returns an object.  |
| <code>tuple(s)</code>              | Converts <code>s</code> to a tuple.  |
| <code>list(s)</code>               | Converts <code>s</code> to a list.   |
| <code>set(s)</code>                | Converts <code>s</code> to a set.  |
| <code>dict(d)</code>               | Creates a dictionary. <code>d</code> must be a sequence of (key,value) tuples.                                 |
| <code>frozenset(s)</code>          | Converts <code>s</code> to a frozen set.   |
| <code>chr(x)</code>                | Converts an integer to a character.  |

# Python Arithmetic Operators

- Assume variable a holds 10 and variable b holds 20

| Operator | Description   | Example   |
|----------|---|---|
| +        | Addition - Adds values on either side of the operator   | $a + b$ will give 30                                |
| -        | Subtraction - Subtracts right hand operand from left hand operand   | $a - b$ will give -10                               |
| *        | Multiplication - Multiplies values on either side of the operator   | $a * b$ will give 200                               |
| /        | Division - Divides left hand operand by right hand operand  | $b / a$ will give 2                                 |
| %        | Modulus - Divides left hand operand by right hand operand and returns remainder   | $b \% a$ will give 0                                |
| **       | Exponent - Performs exponential (power) calculation on operators  | $a^{**}b$ will give 10 to the power 20              |
| //       | Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed. | $9//2$ is equal to 4 and $9.0//2.0$ is equal to 4.0 |

# Python Comparison Operators

- Assume variable a holds 10 and variable b holds 20

| Operator | Description   | Example   |
|----------|---|---|
| ==       | Checks if the value of two operands are equal or not, if yes then condition becomes true.                                       | (a == b) is not true.                             |
| !=       | Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.                      | (a != b) is true.                                 |
| <>       | Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.                      | (a <> b) is true. This is similar to != operator. |
| >        | Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.             | (a > b) is not true.                              |
| <        | Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.                | (a < b) is true.                                  |
| >=       | Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. | (a >= b) is not true.                             |
| <=       | Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.    | (a <= b) is true.                                 |



# Python Assignment Operators

- Assume variable a holds 10 and variable b holds 20

| Operator | Description  | Example  |
|----------|--|--|
| =        | Simple assignment operator, Assigns values from right side operands to left side operand                                     | <code>c = a + b</code> will assign value of <code>a + b</code> into <code>c</code> |
| +=       | Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand                 | <code>c += a</code> is equivalent to <code>c = c + a</code>                        |
| -=       | Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand     | <code>c -= a</code> is equivalent to <code>c = c - a</code>                        |
| *=       | Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand    | <code>c *= a</code> is equivalent to <code>c = c * a</code>                        |
| /=       | Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand         | <code>c /= a</code> is equivalent to <code>c = c / a</code>                        |
| %=       | Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand                   | <code>c %= a</code> is equivalent to <code>c = c % a</code>                        |
| **=      | Exponent AND assignment operator, Performs exponential (power) calculation on operators and assign value to the left operand | <code>c **= a</code> is equivalent to <code>c = c ** a</code>                      |
| //=      | Floor Division and assigns a value, Performs floor division on operators and assign value to the left operand                | <code>c //= a</code> is equivalent to <code>c = c // a</code>                      |

# Python Bitwise Operators

- Assume if  $a = 60$ ; and  $b = 13$ ;
- Now in binary format they will be as follows:

- $a = 0011\ 1100$ ;
- $b = 0000\ 1101$
- $a \& b = 0000\ 1100$
- $a | b = 0011\ 1101$
- $a \wedge b = 0011\ 0001$
- $\sim a = 1100\ 0011$

| Operator | Description   | Example   |
|----------|---|---|
| $\&$     | Binary AND Operator copies a bit to the result if it exists in both operands.   | $(a \& b)$ will give 12 which is 0000 1100  |
| $ $      | Binary OR Operator copies a bit if it exists in either operand.   | $(a   b)$ will give 61 which is 0011 1101   |
| $\wedge$ | Binary XOR Operator copies the bit if it is set in one operand but not both.  | $(a \wedge b)$ will give 49 which is 0011 0001  |
| $\sim$   | Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.   | $(\sim a)$ will give -61 which is 1100 0011 in 2's complement form due to a signed binary number. |
| $\ll$    | Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.   | $a \ll 2$ will give 240 which is 1111 0000  |
| $\gg$    | Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand. | $a \gg 2$ will give 15 which is 0000 1111   |

# Python Logical Operators

- Assume variable a holds 10 and variable b holds 20

| Operator | Description  | Example                |
|----------|--|------------------------|
| and      | Called Logical AND operator. If both the operands are true then then condition becomes true.   | (a and b) is true.     |
| or       | Called Logical OR Operator. If any of the two operands are non zero then then condition becomes true.  | (a or b) is true.      |
| not      | Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false. | not(a and b) is false. |

# Python Membership Operators

- Python has membership operators, which test for membership in a sequence, such as strings, lists, or tuples

| Operator            | Description   | Example  |
|---------------------|---|--|
| <code>in</code>     | Evaluates to true if it finds a variable in the specified sequence and false otherwise.         | <code>x in y</code> , here <code>in</code> results in a 1 if <code>x</code> is a member of sequence <code>y</code> .             |
| <code>not in</code> | Evaluates to true if it does not find a variable in the specified sequence and false otherwise. | <code>x not in y</code> , here <code>not in</code> results in a 1 if <code>x</code> is not a member of sequence <code>y</code> . |

# Example

```
#!/usr/bin/python

a = 10
b = 20
list = [1, 2, 3, 4, 5 ];

if ( a in list ):
    print "Line 1 - a is available in the given list"
else:
    print "Line 1 - a is not available in the given list"

if ( b not in list ):
    print "Line 2 - b is not available in the given list"
else:
    print "Line 2 - b is available in the given list"

a = 2
if ( a in list ):
    print "Line 3 - a is available in the given list"
else:
    print "Line 3 - a is not available in the given list"
```

```
Line 1 - a is not available in the given list
Line 2 - b is not available in the given list
Line 3 - a is available in the given list
```

# Python Operators Precedence

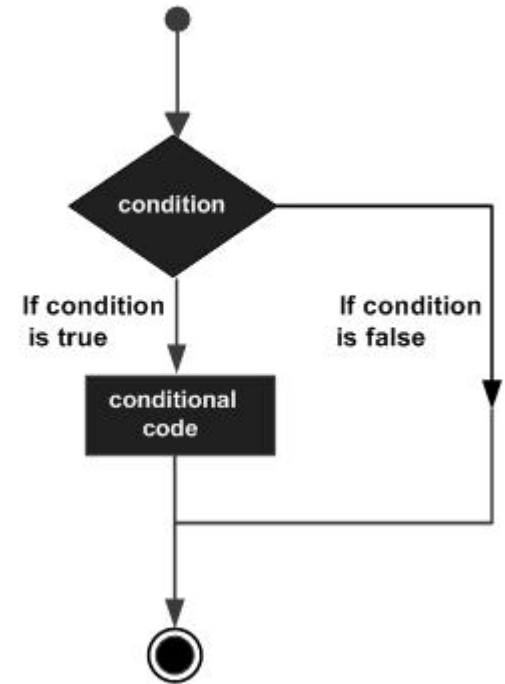
| Operator                    | Description   |
|-----------------------------|---|
| **                          | Exponentiation (raise to the power)   |
| ~ + -                       | Ccomplement, unary plus and minus (method names for the last two are +@ and -@) |
| * / % //                    | Multiply, divide, modulo and floor division                                     |
| + -                         | Addition and subtraction  |
| >> <<                       | Right and left bitwise shift  |
| &                           | Bitwise 'AND'   |
| ^                           | Bitwise exclusive 'OR' and regular 'OR'   |
| <= < > >=                   | Comparison operators  |
| <> == !=                    | Equality operators  |
| = %= /= //= -= += *=<br>**= | Assignment operators  |
| is is not                   | Identity operators  |
| in not in                   | Membership operators  |
| not or and                  | Logical operators   |

# Python Decision Making

| Statement            | Description  |
|----------------------|--|
| if statements        | An <b>if statement</b> consists of a boolean expression followed by one or more statements.  |
| if...else statements | An <b>if statement</b> can be followed by an optional <b>else statement</b> , which executes when the boolean expression is false. |
| nested if statements | You can use one <b>if</b> or <b>else if</b> statement inside another <b>if</b> or <b>else if</b> statement(s).                     |

```
#!/usr/bin/python  
  
var = 100  
  
if ( var == 100 ) : print "Value of expression is 100"  
  
print "Good bye!"
```

```
Value of expression is 100  
Good bye!
```





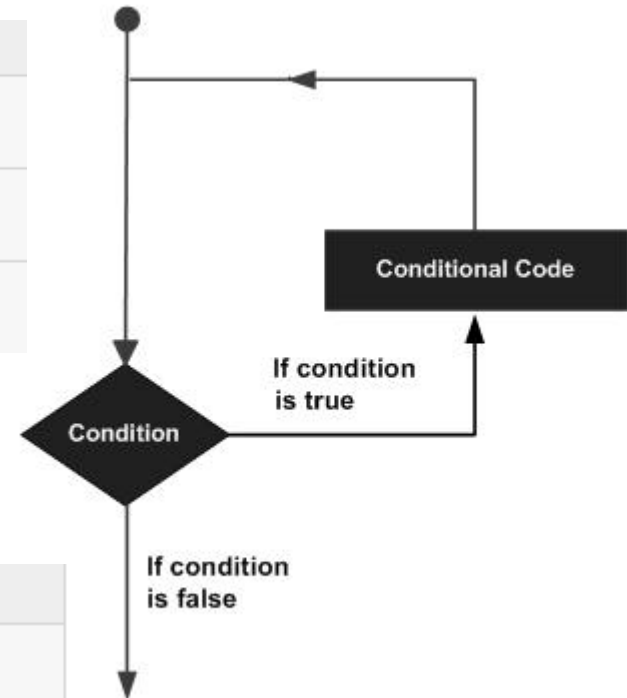
# Conditionals Cont.

- **if** (value is not None) **and** (value == 1):  
    **print** "value equals 1",  
    **print** " more can come in this block"
- **if** (list1 <= list2) **and** (**not** age < 80):  
    **print** "1 = 1, 2 = 2, but 3 <= 7 so its True"
- **if** (job == "millionaire") **or** (state != "dead"):  
    **print** "a suitable husband found"  
**else**:  
    **print** "not suitable"
- **if** ok: **print** "ok"



# Python Loops

| Loop Type    | Description  |
|--------------|--|
| while loop   | Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body. |
| for loop     | Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.                          |
| nested loops | You can use one or more loop inside any another while, for or do..while loop.  |



| Control Statement  | Description   |
|--------------------|---|
| break statement    | Terminates the <b>loop</b> statement and transfers execution to the statement immediately following the loop.                       |
| continue statement | Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.                        |
| pass statement     | The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute. |



# Loops/Iterations

- `sentence = ['Marry', 'had', 'a', 'little', 'lamb']`  
`for word in sentence:`  
    `print word, len(word)`
- `for i in range(10):`  
    `print i`  
`for i in xrange(1000): # does not allocate all initially`  
    `print i`
- `while True:`  
    `pass`
- `for i in xrange(10):`  
    `if i == 3: continue`  
    `if i == 5: break`  
    `print i,`

# pass

- while 1:  
... pass # Busy-wait for keyboard interrupt  
...
- class MyEmptyClass:  
... pass  
...

# range() and xrange()

- range() can construct a numeral list
  - range([start,] stop[, step])

```
1. >>> range(5)
2. [0, 1, 2, 3, 4]
3. >>> range(1,5)
4. [1, 2, 3, 4]
5. >>> range(0,6,2)
6. [0, 2, 4]
```

- xrange() return a generator

```
1. >>> xrange(5)
2. xrange(5)
3. >>> list(xrange(5))
4. [0, 1, 2, 3, 4]
5. >>> xrange(1,5)
6. xrange(1, 5)
7. >>> list(xrange(1,5))
8. [1, 2, 3, 4]
9. >>> xrange(0,6,2)
10. xrange(0, 6, 2)
11. >>> list(xrange(0,6,2))
12. [0, 2, 4]
```

# Difference of range() and xrange()

- range()

```
1. for i in range(0, 100):  
2. print i  
3. for i in xrange(0, 100):  
4. print i
```

```
1. a = range(0,100)  
2. print type(a)  
3. print a  
4. print a[0], a[1]
```

```
1. <type 'list'>  
2. [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24,  
    25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47,  
    48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70  
    , 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 9  
    3, 94, 95, 96, 97, 98, 99]  
3. 0 1
```

# Difference of range() and xrange()

- xrange()

```
1. a = xrange(0,100)
2. print type(a)
3. print a
4. print a[0], a[1]
```

```
1. <type 'xrange'>
2. xrange(100)
3. 0 1
```

# Defining a Function

- Begin with the keyword **def** followed by the function name and parentheses ( **( )** ).
- Any input parameters or arguments should be placed within these parentheses.
- The first statement of a function can be an optional statement - the documentation string of the *function* or *docstring*.
- The code block within every function starts with a colon (**:**) and is indented.
- The statement `return [expression]` exits a function, optionally passing back an expression to the caller.
  - A return statement with no arguments is the same as `return None`.

# Defining a Function

```
def functionname( parameters ):  
    "function_docstring"  
    function_suite  
    return [expression]
```

```
def printme( str ):  
    "This prints a passed string into this function"  
    print str  
    return
```

```
#!/usr/bin/python  
  
# Function definition is here  
def printme( str ):  
    "This prints a passed string into this function"  
    print str;  
    return;  
  
# Now you can call printme function  
printme("I'm first call to user defined function!");  
printme("Again second call to the same function");
```

```
I'm first call to user defined function!  
Again second call to the same function
```





# Functions

- `def print_hello():# returns nothing`  
`print "hello"`
- `def gcd(m, n):`  
`if n == 0:`  
 `return m # returns m`  
`else:`  
 `return gcd(n, m % n)`
- `def has_args(arg1,arg2=['e', 0]):`  
 `num = arg1 + 4`  
 `mylist = arg2 + ['a',7]`  
 `return [num, mylist]`  
`has_args(5.16,[1,'b']) # returns [9.16,[1, 'b','a',7]`

# Pass by value

```
#!/usr/bin/python

# Function definition is here
def changeme( mylist ):
    "This changes a passed list into this function"
    mylist.append([1,2,3,4]);
    print "Values inside the function: ", mylist
    return

# Now you can call changeme function
mylist = [10,20,30];
changeme( mylist );
print "Values outside the function: ", mylist
```

```
Values inside the function: [10, 20, 30, [1, 2, 3, 4]]
Values outside the function: [10, 20, 30, [1, 2, 3, 4]]
```

# Pass by reference

```
#!/usr/bin/python

# Function definition is here
def changeme( mylist ):
    "This changes a passed list into this function"
    mylist = [1,2,3,4]; # This would assign new reference in mylist
    print "Values inside the function: ", mylist
    return

# Now you can call changeme function
mylist = [10,20,30];
changeme( mylist );
print "Values outside the function: ", mylist
```

```
Values inside the function:  [1, 2, 3, 4]
Values outside the function:  [10, 20, 30]
```

# Default arguments

```
#!/usr/bin/python

# Function definition is here
def printinfo( name, age = 35 ):
    "This prints a passed info into this function"
    print "Name: ", name;
    print "Age ", age;
    return;

# Now you can call printinfo function
printinfo( age=50, name="miki" );
printinfo( name="miki" );
```

```
Name:  miki
Age   50
Name:  miki
Age   35
```

# Variable-length arguments

- You may need to process a function for more arguments than you specified while defining the function.

```
def functionname([formal_args,] *var_args_tuple ):
    "function_docstring"
    function_suite
    return [expression]
```

```
#!/usr/bin/python

# Function definition is here
def printinfo( arg1, *vartuple ):
    "This prints a variable passed arguments"
    print "Output is: "
    print arg1
    for var in vartuple:
        print var
    return;

# Now you can call printinfo function
printinfo( 10 );
printinfo( 70, 60, 50 );
```

```
Output is:
10
Output is:
70
60
50
```

# The *Anonymous* Functions

- You can use the *lambda* keyword to create small anonymous functions.
  - These functions are called anonymous because they are not declared in the standard manner by using the *def* keyword.
- The syntax of *lambda* functions contains only a single statement,

```
lambda [arg1 [,arg2,.....argn]]:expression
```

```
#!/usr/bin/python

# Function definition is here
sum = lambda arg1, arg2: arg1 + arg2;

# Now you can call sum as a function
print "Value of total : ", sum( 10, 20 )
print "Value of total : ", sum( 20, 20 )
```

```
Value of total : 30
Value of total : 40
```

# Overview of OOP Terminology

- **Class:** A user-defined prototype for an object that defines a set of attributes that characterize any object of the class.
  - The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.
- **Class variable:** A variable that is shared by all instances of a class.
  - Class variables are defined within a class but outside any of the class's methods.
  - Class variables aren't used as frequently as instance variables are.
- **Data member:** A class variable or instance variable that holds data associated with a class and its objects.
- **Instance variable:** A variable that is defined inside a method and belongs only to the current instance of a class.

# Creating Classes

- The *class* statement creates a new class definition.

```
class ClassName:  
    'Optional class documentation string'  
    class_suite
```

- The class has a documentation string, which can be accessed via *ClassName.\_\_doc\_\_*.
- The *class\_suite* consists of all the component statements defining class members, data attributes and functions.



# EXAMPLE

- The variable *empCount* is a **class variable** whose value would be shared among all instances of a this class.
  - This can be accessed as *Employee.empCount* from inside the class or outside the class.
- The first method `__init__()` is a special method, which is called class **constructor** or **initialization method** that Python calls when you create a new instance of this class.
- You declare other class methods like normal functions with the exception that the first argument to each method is *self*.
  - Python adds the *self* argument to the list for you; you don't need to include it when you call the methods.

```
class Employee:
    'Common base class for all employees'
    empCount = 0

    def __init__(self, name, salary):
        self.name = name
        self.salary = salary
        Employee.empCount += 1

    def displayCount(self):
        print "Total Employee %d" % Employee.empCount

    def displayEmployee(self):
        print "Name : ", self.name, ", Salary: ", self.salary
```

# Creating instance objects

- To create instances of a class, you call the class using class name and pass in whatever arguments its `__init__` method accepts.

```
"This would create first object of Employee class"  
emp1 = Employee("Zara", 2000)  
"This would create second object of Employee class"  
emp2 = Employee("Manni", 5000)
```

- Accessing attributes

```
emp1.displayEmployee()  
emp2.displayEmployee()  
print "Total Employee %d" % Employee.empCount
```

# Example

```
#!/usr/bin/python

class Employee:
    'Common base class for all employees'
    empCount = 0

    def __init__(self, name, salary):
        self.name = name
        self.salary = salary
        Employee.empCount += 1

    def displayCount(self):
        print "Total Employee %d" % Employee.empCount

    def displayEmployee(self):
        print "Name : ", self.name, ", Salary: ", self.salary

"This would create first object of Employee class"
emp1 = Employee("Zara", 2000)
"This would create second object of Employee class"
emp2 = Employee("Manni", 5000)
emp1.displayEmployee()
emp2.displayEmployee()
print "Total Employee %d" % Employee.empCount
```

```
Name : Zara ,Salary: 2000
Name : Manni ,Salary: 5000
Total Employee 2
```





# Python semantics

- Each statement has its own semantics, the def statement doesn't get executed immediately like other statements
- Python uses duck typing, or latent typing
  - This means you can just declare “somevariable = 69” don't actually have to declare a type
  - Allows for polymorphism without inheritance
  - print “somevariable = “ + tostring(somevariable)”
- strong typing , can't do operations on objects not defined without explicitly asking the operation to be done



# Python Syntax

- Python uses indentation and/or whitespace to delimit statement blocks rather than keywords or braces

- **if** `__name__ == "__main__":`

**print** "Salve Mundo"

*# if no comma (,) at end '\n' is auto-included*

## *CONDITIONALS*

- **if** (i == 1): do\_something1()  
**elif** (i == 2): do\_something2()  
**elif** (i == 3): do\_something3()  
**else**: do\_something4()



# Exception handling

- `try:`  
    `f = open("file.txt")`  
`except IOError:`  
    `print "Could not open"`  
`else:`  
    `f.close()`
- `a = [1,2,3]`  
`try:`  
    `a[7] = 0`  
`except (IndexError,TypeError):`  
    `print "IndexError caught"`  
`except Exception, e:`  
    `print "Exception: ", e`  
`except: # catch everything`  
  
    `print "Unexpected:"`  
    `print sys.exc_info()[0]`  
    `raise # re-throw caught exception`
- `try:`  
    `a[7] = 0`  
`finally:`  
    `print "Will run regardless"`
- Easily make your own exceptions:  
`class myException(except)`  
    `def __init__(self,msg):`  
        `self.msg = msg`  
    `def __str__(self):`  
        `return repr(self.msg)`

# Classes

- Creating Classes: The *class* statement creates a new class definition.
- The name of the class immediately follows the keyword *class* followed by a colon as follows:

```
class ClassName:  
    'Optional class documentation string'  
    class_suite
```

```
class Employee:  
    'Common base class for all employees'  
    empCount = 0  
  
    def __init__(self, name, salary):  
        self.name = name  
        self.salary = salary  
        Employee.empCount += 1  
  
    def displayCount(self):  
        print "Total Employee %d" % Employee.empCount  
  
    def displayEmployee(self):  
        print "Name : ", self.name, ", Salary: ", self.salary
```



# Classes

- The variable *empCount* is a class variable whose value would be shared among all instances of a this class.
- The first method `__init__()` is a special method which is called class constructor or initialization method.
  - Python calls when you create a new instance of this class.
- Python adds the *self* argument to the list for you; you don't need to include it when you call the methods.

```
class Employee:
    'Common base class for all employees'
    empCount = 0

    def __init__(self, name, salary):
        self.name = name
        self.salary = salary
        Employee.empCount += 1

    def displayCount(self):
        print "Total Employee %d" % Employee.empCount

    def displayEmployee(self):
        print "Name : ", self.name, ", Salary: ", self.salary
```



# Classes

```
class MyVector: """A simple vector class."""
```

```
    num_created = 0
```

```
    def __init__(self,x=0,y=0):
```

```
        self.__x = x
```

```
        self.__y = y
```

```
        MyVector.num_created += 1
```

```
    def get_size(self):
```

```
        return self.__x+self.__y
```

```
    @staticmethod
```

```
    def get_num_created
```

```
        return MyVector.num_created
```

```
#USAGE OF CLASS MyVector
```

```
print MyVector.num_created
```

```
v = MyVector()
```

```
w = MyVector(0.23,0.98)
```

```
print w.get_size()
```

```
bool = isinstance(v, MyVector)
```

Output:

0

1.21



# I/O

```
import os
print os.getcwd() #get "."
os.chdir('..')
import glob # file globbing
lst = glob.glob('*.*txt') # get list of files
import shutil # mngmt tasks
shutil.copyfile('a.py', 'a.bak')

import pickle # serialization logic
ages = {"ron":18,"ted":21}
pickle.dump(ages,fout)
# serialize the map into a writable file
ages = pickle.load(fin)
# deserialize map from a readable
file
```

```
# read binary records from a file
import *
fin = None
try:
    fin = open("input.bin","rb")
    s = f.read(8)#easy to read in
    while (len(s) == 8):
        x,y,z = unpack(">HH<L", s)
        print "Read record: " \
            "%04x %04x %08x"%(x,y,z)
        s = f.read(8)
except IOError:
    pass
if fin: fin.close()
```



# Threading in Python

```
import threading
```

```
theVar = 1
```

```
class MyThread ( threading.Thread ):
```

```
    def run ( self ):
```

```
        global theVar
```

```
        print 'This is thread ' + \
```

```
        str ( theVar ) + ' speaking.'
```

```
        print 'Hello and good bye.'
```

```
        theVar = theVar + 1
```

```
for x in xrange ( 10 ):
```

```
    MyThread().start()
```

```
C:\WINDOWS\system32\cmd.exe
C:\Documents and Settings\far
This is thread 1 speaking.
Hello and good bye.
This is thread 2 speaking.
Hello and good bye.
This is thread 3 speaking.
Hello and good bye.
This is thread 4 speaking.
Hello and good bye.
This is thread 5 speaking.
Hello and good bye.
This is thread 6 speaking.
Hello and good bye.
This is thread 7 speaking.
Hello and good bye.
This is thread 8 speaking.
Hello and good bye.
This is thread 9 speaking.
Hello and good bye.
This is thread 10 speaking.
Hello and good bye.
C:\Documents and Settings\far
```



# So what does Python have to do with Internet and web programming?

- Jython
  - Jython use Java to implement Python.
  - Using Jython, Python can communicate with .NET each other.
- IronPython(.NET ,written in C#)
  - IronPython is used in .NET platform
  - IronPython can be integrated in C #

# So what does Python have to do with Internet and web programming?

- Libraries – ftplib, snmplib, uuidlib, smtpd, urlparse, SimpleHTTPServer, cgi, telnetlib, cookielib, xmlrpclib, SimpleXMLRPCServer, DocXMLRPCServer
- Zope(application server), PyBloxsom(blogger), MoinMoin(wiki), Trac(enhanced wiki and tracking system), and Bittorrent (6 no, but prior versions yes)



# Python Interpreters

- <http://www.python.org/download/>
- <http://pyaiml.sourceforge.net/>
- <http://www.py2exe.org/>
- <http://www.activestate.com/Products/activepython/>
- <http://www.wingware.com/>
- <http://pythonide.blogspot.com/>
- Many more...



# Python on your systems

- Its easy! Go to <http://www.python.org/download/>
- Download your architecture binary, or source
- Install, make, build whatever you need to do... plenty of info on installation in readmes
- Make your first program! (a simple one like the hello world one will do just fine)
- Two ways of running python code. Either in an interpreter or in a file ran as an executable





# Running Python

- Windows XP – double click the icon or call it from the command line as such:



```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\farrin>cd Desktop

C:\Documents and Settings\farrin\Desktop>test.py
Hello World!

C:\Documents and Settings\farrin\Desktop>
```



# Python Interpreter

```
Python (command line)
Python 2.5.2 (r252:60911, Feb 21 2008, 13:11:45) [MSC v.1310 32 bit (Intel)] on
win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print 'hello world'
hello world
>>> x = 'roses'
>>> y = 12
>>> Roy_G_Biv = [('red', 'orange', 'yellow'), 'green', ('blue', ['indigo', 'violet'])
]
>>> MyAwesomeVar = [x, y, Roy_G_Biv]
>>> print MyAwesomeVar
['roses', 12, [('red', 'orange', 'yellow'), 'green', ('blue', ['indigo', 'violet
'])]]
>>> print MyAwesomeVar[0]+' are '+MyAwesomeVar[2:3][0][0][0]
roses are red
>>> print MyAwesomeVar[2:3][0][2][1][1]+'s are '+MyAwesomeVar[2:3][0][2][0]
violets are blue
>>> print str(MyAwesomeVar[1])+' '+MyAwesomeVar[0]+' for my love.'
12 roses for my love.
>>> dict = {'place': 'mantle', 'where': 'above', 'myLove': True}
>>> if(dict["myLove"]): print 'the ' +dict["place"]+' I put them '+dict["where"]

...
the mantle I put them above
>>>
```



# Python for the future

- Python 3.0
  - Will not be Backwards compatible, they are attempting to fix “perceived” security flaws.
  - Print statement will become a print function.
  - All text strings will be unicode.
  - Support of optional function annotation, that can be used for informal type declarations and other purposes.



# Bibliography

- <http://it.metr.ou.edu/byteofpython/features-of-python.html>
- <http://codesyntax.netfirms.com/lang-python.htm>
- <http://www.python.org/>
- Sebesta, Robert W., Concepts of Programming Languages: 8th ed. 2007
- <http://www.python.org/~guido/>