Python

The Simplest Python Program

```
#!/usr/bin/python

print "Hello, Python!”
```

To compile and run:

```
$ python hello.py
Hello, Python!
```

About Python

Python is an interpreted, interactive, object-oriented language:

- interpreted—processed at runtime by the interpreter. Python code does not need to be pre-compiled before running (just like Perl or PhP)
- interactive—there is a Python prompt where commands can be typed
- object-oriented—Python supports code within objects.

Installing Python

To find out whether Python is installed on your system, type `python` from the command line:

```
$ python
Python 2.7.6 (default, Sep  9 2014, 15:04:36)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.39)] on darwin
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

This also drops you into Python prompt mode.

Python can be downloaded from www.python.org.

For MacOS X, download from http://www.python.org/download/mac/. For Windows, download from http://www.python.org/download/, and follow the link for the Windows installer `python-XYZ.msi` file, where XYZ is the version. The

Windows system must support Microsoft Installer 2.0. Run the downloaded file. This brings up the Python install wizard, which is really easy to use.

Command Line Options

Option	Description
`-d`	with debug
`-O`	generate optimized bytecode (*.pyo files)
`-S`	do not run import site to look for Python paths on startup
`-v`	verbose output
`-X`	disable class-based build-in exceptions
`-c cmd`	run Python script sent in as cmd string
`file`	run Python script from given file

Interactive vs. Script Mode Programming

Interactive prompt:

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

Script mode:

```
$ python hello.py
Hello, Python!
```

Python Identifiers

Identifiers are names for variables, functions, classes, modules, or other objects. Identifiers can start with uppercase letters A…Z, lowercase letters a…z, underscores _, and digits 0…9. Identifiers cannot contain characters such as @, $, or %. Python is case sensitive.

Naming conventions in Python:
classes—start with an uppercase letter
other identifiers—lowercase letter
private identifiers—start with a single leading underscore
strongly private identifiers—start with two leading underscores

language-defined special names—end with two trailing underscores

Reserved words:

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

Proper Indentation

Python relies on strict indentation to delimit blocks of code, such as class and function definitions, or flow control.

Strings

Strings are sequences of characters enclosed in quotes.

Example:

```
#!/usr/bin/python

greeting='Hello World!'
my_name="Adriana WISE"

print "greeting[0:6]: ", greeting[0:6]
print "my_name[0:8]: ", my_name[0:8]
```

Output:

```
$ python strings.py
greeting[0:6]:  Hello
my_name[0:8]:  Adriana
```

Lists

Lists are sequences of comma-separated values within square brackets, not necessarily of the same type.

Example:

```
#!/usr/bin/python

names=['Lyle PUENTE', 'Tyler JOSEPH', 'Josh DUN']
numbers= [1, 2, 3, 4, 5, 6, 7 ];

print "names[0]: ", names[0]
print "numbers[0:5]: ", numbers[0:5]
```

Output:

```
$ python lists.py
names[0]:  Lyle PUENTE
numbers[0:5]:  [1, 2, 3, 4, 5]
```

Tuples

Unlike lists, tuples are immutable (cannot be changed). Their syntax includes parentheses, not square brackets.

Dictionaries

A dictionary is like a hash table. It consists of a sequence of key, value pairs. Keys are immutable (just like in a database).

```
#!/usr/bin/python

dict = {'Name':'Lyle PUENTE', 'Age':53, 'Num_Albums':6}

print "dict['Name']=", dict['Name']
print "dict['Age']=", dict['Age']
```

Output:

```
$ python dictionary.py
dict['Name']= Lyle PUENTE
dict['Age']= 53
```

Python provides a series of dictionary functions (functions that take a dictionary value as an argument) and methods (functions that operate on a dictionary object).

Function	Description
`cmp(dict1, dict2)`	Compares 2 dictionaries element by element. Returns boolean.
`len(dict)`	Returns # of elements of a dictionary.
`str(dict)`	Returns a printable string representation of a dictionary.
`type(variable)`	Returns the type of the argument variable.

The following table gives the dictionary methods:

Method	Description
`dict.clear()`	Clears dictionary object **`dict`**.
`dict.copy()`	Returns a copy of object **`dict`**.
`dict.fromkeys()`	Creates a dictionary with keys from sequence, and values set to values.
`dict.get(key, default=None)`	Returns values associated with **`key`**, or none if key not in dictionary.
`dict.has_key(key)`	Returns true if **`key`** in dictionary.
`dict.items()`	Returns all **`(key, value)`** pairs.
`dict.keys()`	Returns a list of all keys of **`dict`**.
`dict.setdefault(key, default=None)`	Same as **`get()`**, but will set **`dict[key]=default`** if **`key`** not in **`dict`**.

Method	Description
dict.update(dict2)	Adds **(key, value)** pairs of **dict2** to **dict**.
dict.values()	Returns a list of all values of **dict**.

Example:

```
#!/usr/bin/python

dict = {'Name':'Lyle PUENTE', 'Age':53, 'Num_Albums':6};

print "dict['Name']=", dict['Name']
print "dict['Age']=", dict['Age']
print dict.items();
print dict.keys();
```

Output:

```
$ python dictionary.py
dict['Name']= Lyle PUENTE
dict['Age']= 53
[('Age', 53), ('Name', 'Lyle PUENTE'), ('Num_Albums', 6)]
['Age', 'Name', 'Num_Albums']
```

Decision Statements

An `if-else` statement evaluates the truth value of a Boolean expression and executes the if branch on TRUE, otherwise the else branch on FALSE.

Example:

```
#!/usr/bin/python

my_string='Adriana WISE'

if (my_string=='Adriana WISE'):
        print "My name is %s" % my_string

print "Good bye!"
```

Output:

```
$ python if.py
My name is Adriana WISE
Good bye!
```

The `for` Loop

The `for` loop executes a statement or a block of statements a fixed number of times, as stated in the `for` expression.

Example:

```
#!/usr/bin/python

my_name='Adriana WISE'

for i in range(0, 10):
        print my_name
```

Output:

```
$ python for.py
Adriana WISE
Adriana WISE
Adriana WISE
Adriana WISE
Adriana WISE
Adriana WISE
Adriana WISE
Adriana WISE
Adriana WISE
Adriana WISE
```

The `while` Loop

The `while` loop executes a statement or a block of statements for as many iterations as the condition set in the Boolean expression from the while statement remains TRUE.

Example:

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

```
my_name='Adriana WISE'

i=0
while (i<10):
        print my_name
        i+=1
```

Output:

```
$ python while.py
Adriana WISE
Adriana WISE
Adriana WISE
Adriana WISE
Adriana WISE
Adriana WISE
Adriana WISE
Adriana WISE
Adriana WISE
Adriana WISE
```

Functions

Like every other language, Python provides **built-in functions** and **user-defined functions**. Here are some of the rules for designing user-defined functions:

- Function blocks begin with the keyword `def` followed by the function name and parentheses.
- Function **input parameters** (a.k.a. **arguments**) are listed comma-separated 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 a return value to the caller (another function).

Syntax:

```
def function_name(parameters):
   "function_docstring"
```

```
    function_suite
    return [expression]
```

Example:

```
#!/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
my_string="My name is Adriana WISE."
printme(my_string)
```

Output:

```
$ python function.py
My name is Adriana WISE.
```

Pass by Reference vs. Pass by Value

In Python, all arguments are passed by reference. Any modification to the argument value made within the function will reflect in the caller. For example, if in our `printme()` function we changed the value of `num`, resetting it to a new value, the caller will reflect this new value. This behavior is not true for call-by-value arguments in languages supporting that (such as C, C++, Pascal etc.). It is *not true for immutable types* such as numeric, string in Python, either.

Example:

```
#!/usr/bin/python

# Function definition is here
def change(original_list):
        "This changes a list passed into this function"
        original_list+=['Josh DUN']
        return

original_list=['Lyle PUENTE', 'Tyler JOSEPH']
```

```
print original_list
change(original_list)
print original_list
```

Output:

```
$ python function2.py
['Lyle PUENTE', 'Tyler JOSEPH']
['Lyle PUENTE', 'Tyler JOSEPH', 'Josh DUN’]
```

Variable Argument Lists

Python supports variable argument lists. The following example shows a function with two arguments, of which the second one has variable length.

Example:

```
#!/usr/bin/python

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

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

Output:

```
$ python vararglists.py
Output is:
arg1= 10
Output is:
arg1= 70
60
50
```

Lambda Functions

These functions are also called **anonymous** because they are not declared in the standard manner by using the `def` keyword. Instead, the `lambda` keyword is used to create small anonymous functions.

- lambda functions can take any number of arguments, but their body contains one line (one expression), whose value they return
- lambda functions cannot access variables other than those in their argument list, or those in the global namespace

Syntax:

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

Example:

```
#!/usr/bin/python

sum=lambda arg1, arg2: arg1 + arg2;

print "Sum=", sum(10, 20)
print "Sum=", sum(20, 20)

#Alternate function definition
def sum(arg1, arg2):
    "This function returns the sum of its arguments"
    s=arg1+arg2
    return s
```

Output:

```
$ python lambda.py
Sum= 30
Sum= 40
```

Global vs. Local Variables

Like in other languages, Python recognizes a **global scope**, meaning that variables declared globally are visible to every function; and a **local scope**, meaning that variables declared local to a function or a block of code are only visible in that scope.

Example:

```
#!/usr/bin/python

total=0; #This is a global variable.

#Function definition
def sum(arg1, arg2):
        "Add both the parameters and return them."
        total=arg1+arg2;
        print "Inside the function, local variable total=",
total
        return total;

sum(10, 20);
print "Outside the function, global variable total=", total
```

Output:

```
$ python globalvslocal.py
Inside the function, local variable total= 30
Outside the function, global variable total= 0
```

Python Modules

A Python **module** allows functions or classes to be defined in a separate file, which can be imported in the main program. This separation makes it easier to maintain large code categorized by functionality and classes.

Below is a simple example, consisting of a function defined in a separate file. The file is then imported as a module in the main program. By so doing, the function is automatically known, and can be called by, the main program.

Example:

File `name_mod.py`, importable as module `name_mod`:

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

File `main_prog.py`, which imports `name_mod`, and which will be executed:

```
#!/usr/bin/python

# Import module support
import name_mod

# Now you can call defined function that module as follows
name_mod.print_func("Adriana WISE”)
```

Output:

```
$ python main_prog.py
Hello, Adriana WISE
```

A main program may import only part of the attributes defined in a Python module, and not the entire module. The list of attributes specified in the import statement is then included in the global symbol table of the importing module (or main program).

The `dir()` function returns all attributes defined in a module.

Syntax:

```
from mod_name import name1[, name2, …, nameN]
```

Example:

```
#!/usr/bin/python

# Import module support
import name_mod as module
from module import hello_func

content=dir(module)
print content

hello_func("Adriana WISE")
```

Output:

```
$ python main_prog.py
```

```
['__builtins__', '__doc__', '__file__', '__name__',
'__package__', 'bye_func', 'hello_func']
Hello, Adriana WISE
```

However, if we wanted to access the function `module.bye_func()` from `name_mod` imported as `module`, we would get an error:

Source:

```
#!/usr/bin/python

# Import module name_mod
import name_mod as module
from name_mod import hello_func

content=dir(module)
print content

hello_func("Adriana WISE")
bye_func("Lyle PUENTE”)
```

Output:

```
$ python main_prog.py
['__builtins__', '__doc__', '__file__', '__name__',
'__package__', 'bye_func', 'hello_func']
Hello, Adriana WISE
Traceback (most recent call last):
  File "main_prog.py", line 11, in <module>
    bye_func("Lyle PUENTE")
NameError: name 'bye_func' is not defined
```

File I/O

Writing to standard output (terminal) is done with the `print` function:

```
#!/usr/bin/python

print "My name is Adriana WISE.”
```

Output:

```
$ python print.py
```

```
My name is Adriana WISE.
```

(Same old, same old.)

Reading from standard output is done with two Python built-in functions, `raw_input()` and `input()`. The `raw_input()` function reads one line from the command line and returns the input as string. The `input()` function interprets the input expression or variable to the appropriate type.

Example 1:

```
#!/usr/bin/python

str = raw_input("Enter your name: ");
print "Hello,", str
```

Output:

```
$ python input.py
Enter your name: Adriana WISE
Hello, Adriana WISE
```

Example 2:

```
#!/usr/bin/python

str = input("Enter your input: ");
print "Received input is : ", str
```

Output:

```
$ python raw_input.py
Enter your input: [x*5 for x in range(2, 10)]
Received input is :  [10, 15, 20, 25, 30, 35, 40, 45]

$ python raw_input.py
Enter your input: [x*5 for x in range(2, 10, 2)]
Received input is :  [10, 20, 30, 40]
```

File I/O is done via a **file object**. To open a file for reading or writing, Python provides the `open()` function, called with the following parameters:
file_name—a string value with the name of the file

access_mode—the mode of file opening: read, write, append etc. A complete list of the file access modes is given below
buffering—I/O can be unbuffered (with an arg value of 0) or buffered (an arg value of 1).

Mode	Description
r	Open file for reading.
rb	Open file for reading in binary format.
r+	Open file for reading and writing.
rb+	Open file for reading and writing in binary format.
w	Opens a file for writing, overwrites if file exists.
wb	Opens file for writing in binary format.
w+	Opens file for writing and reading.
wb+	Opens file for writing and reading in binary format.
a	Opens a file for appending. Creates a new file if file does not exist.
ab	Opens a file for appending in binary format.
a+	Opens file for appending and reading.
ab+	Opens file for appending and reading in binary format.

The file object attributes are shown in the following table.

Attribute	Description
file.closed	TRUE if file is closed, FALSE otherwise.
file.mode	Returns access mode for file.
file.name	Returns name of file.
file.softspace	FALSE if space explicitly with print, TRUE otherwise.

Example:

```
#!/usr/bin/python

# Open a file
fo=open("text.txt", "wb")
print "Name of the file: ", fo.name
print "Closed or not : ", fo.closed
print "Opening mode : ", fo.mode
print "Softspace flag : ", fo.softspace
```

Output:

```
$ python fileio.py
Name of the file:  text.txt
Closed or not :  False
Opening mode :  wb
Softspace flag :  0
```

The `close()` function closes a file.

Example:

```
#!/usr/bin/python

# Open a file
fo=open("text.txt", "wb")
print "Name of the file: ", fo.name

# Close opened file
fo.close()
```

Output:

```
$ python close.py
Name of the file:  text.txt
```

The file object can be read or written with one of the `read()` or `write()` functions.

Example 1:

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

```
# Open a file
fo=open("foo.txt", "wb")
fo.write( "Python is a great language.\nYeah its great!!\n");

# Close opened file
fo.close()
```

Output:

```
$ more foo.txt
Python is a great language.
Yeah its great!!
```

Example 2:

```
#!/usr/bin/python

# Open a file
fo=open("foo.txt", "r+")
str=fo.read(10);
print "Read string is : ", str
# Close opened file
fo.close()
```

Output:

```
$ python read.py
Read string is :  Python is
```

Classes

Class: A user-defined prototype for an object, defining a set of attributes that characterize any object of the class. The attributes are **data members** and **methods**, accessed via dot notation. Terminology:

- **Class variable:** A variable that is shared by all instances of a class.
- **Data member:** A class variable that holds data associated with a class.
- **Function overloading:** The assignment of more than one behavior to a particular function. The operation performed varies with the types of arguments to the function.
- **Instance variable:** A variable that is defined inside a method and belongs only to the current instance of a class.

- **Inheritance:** The transfer of the characteristics of a class to other classes that are derived from it.
- **Instance:** An object from a class.
- **Instantiation:** The creation of an object, as an "instance" of a class.
- **Method :** A class function.
- **Object:** An instance of the class. An object comprises both data members (class variables and instance variables) and methods.
- **Operator overloading:** The assignment of more than one function to a particular operator.

Example:

```
class Musician:
    'Common base class for all musicians'
    musiciansCount = 0

    def __init__(self, name, age):
        self.name = name
        self.age = age
        Musician.musiciansCount += 1

    def displayCount(self):
        print "Number of musicians %d" % Musician.musiciansCount

    def displayMusician(self):
        print "Name : ", self.name,  ", Age: ", self.age
```

Class instantiation:

```
musician1=Musician("Lyle PUENTE", 53)
musician2=Musician("Tyler JOSEPH", 26)
musician3=Musician("Josh DUN", 27)
```

The following functions can be used to access attribute information:

`getattr(obj, name[, default])` : accesses the attribute of object.

`hasattr(obj, name)` : checks if an attribute exists or not.

`setattr(obj, name, value)` : sets an attribute's value; creates attribute if it does not exist.

`delattr(obj, name)` : deletes an attribute.

Every Python class has a number of built-in attributes, accessible with the dot operator like all other attributes:

__dict__: Dictionary containing the class's namespace.
__doc__: Class documentation string or none, if undefined.
__name__: Class name.
__module__: Module name in which the class is defined. This attribute is "__main__" in interactive mode.
__bases__: A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.

Example:

```
#!/usr/bin/python

class Musician:
        'Common base class for all musicians'
        musiciansCount = 0

        def __init__(self, name, age):
                self.name = name
                self.age = age
                Musician.musiciansCount += 1

        def displayCount(self):
                print "Total musicians %d" %
Musician.musiciansCount

        def displayMusician(self):
                print "Name : ", self.name,  ", Age: ", self.age

                print "Musician.__doc__:", Musician.__doc__
                print "Musician.__name__:", Musician.__name__
                print "Musician.__module__:",
Musician.__module__
                print "Musician.__bases__:", Musician.__bases__
                print "Musician.__dict__:", Musician.__dict__

musician1=Musician("Lyle PUENTE", 53)
musician2=Musician("Tyler JOSEPH", 26)

musician1.displayMusician()
musician2.displayMusician()
```

Output:

```
$ python class.py
Name :  Lyle PUENTE , Age:  53
Musician.__doc__: Common base class for all musicians
Musician.__name__: Musician
Musician.__module__: __main__
Musician.__bases__: ()
Musician.__dict__: {'musiciansCount': 2, '__module__':
'__main__', 'displayCount': <function displayCount at
0x103a01aa0>, 'displayMusician': <function displayMusician at
0x103a01410>, '__doc__': 'Common base class for all musicians',
'__init__': <function __init__ at 0x1039fec80>}
Name :  Tyler JOSEPH , Age:  26
Musician.__doc__: Common base class for all musicians
Musician.__name__: Musician
Musician.__module__: __main__
Musician.__bases__: ()
Musician.__dict__: {'musiciansCount': 2, '__module__':
'__main__', 'displayCount': <function displayCount at
0x103a01aa0>, 'displayMusician': <function displayMusician at
0x103a01410>, '__doc__': 'Common base class for all musicians',
'__init__': <function __init__ at 0x1039fec80>}
```

Regular Expressions

Like Perl, Python has built-in functions to deal with finding patterns into strings, a.k.a. **regular expressions**. These functions are `match()` and `search()`. The module `re` (**r**egular **e**xpressions) provides full support for Perl-like regular expressions in Python. The `re` module raises the exception `re.error` if an error occurs while compiling or using a regular expression.

Syntax:

```
re.match(pattern, string, flags=0)
```

The arguments are:
`pattern`—the regular expression (pattern) to be found and matched in the string
`string`—the string the pattern is searched into
`flags`—modifiers which can be combined with bitwise or |.

Metacharacters	Meaning
-	range
.	matches any character, except a newline character. If the DOTALL flag was specified, it also matches a newline
^	complements a characters class, i.e. items not in the class, e.g. [^5]
^	matches the regex at the start of a string. In MULTILINE mode also matches immediately after each newline
$	matches the regex at the end of a string
*	greedy repetition (matches 0 or more times)
+	matches 1 or more times
?	matches 1 or 0 times
{}	{m, n} at least m, and at most n repetitions
[]	specify a character class, meaning a set of characters to match, e.g. [abc], [a-c]
\	escape character (to match metacharacters, for example)
|	A|B where A, B are regular expressions (patterns) creates a new regex (a new pattern) that will match either A or B
()	matches a regex indicated within the (), and is used for applying other qualifiers to the regex within the ()

The following table shows some of the methods of the match object:

Method	Meaning
group()	return the entire string matched by the regex
start()	return the starting position of the match
end()	return the ending position of the match

Method	Meaning
span()	return a tuple (start, end) containing the pair starting, ending position of the match

Example:

```
#!/usr/bin/python

import re

my_string='Lyle PUENTE'

regex='(.yle)*'

matchObj=re.match(regex, my_string, re.I|re.M)

if matchObj:
        print matchObj
        print matchObj.group()
else:
        print 'No match!'
```

Output:

```
$ python regex1.py
<_sre.SRE_Match object at 0x1033c3a80>
Lyle
```

To find all the matches of a pattern into a string, Python provides two built-in methods for the `re` object, `re.findall(regex, string)` and `re.finditer(regex, string)`.

Example 1:

```
#!/usr/bin/python

import re

my_string='Lyle PUENTE and Tyler JOSEPH'

regex='(.yle.)'
```

```
matchObj2=re.findall(regex, my_string)

if matchObj2:
        print matchObj2
else:
        print 'No match!'
```

Output:

```
$ python regex1.py
['Lyle ', 'Tyler']
```

Example 2:

```
#!/usr/bin/python

import re

my_string='Lyle PUENTE and Tyler JOSEPH'

regex='(.yle.)'

for m in re.finditer(regex, my_string):
        print m.group()
```

Output:

```
$ python regex2.py
Lyle
Tyler
```