Understanding PYTHON

By,
Anand Roy Choudhury
CSE Department,
IIT Bombay.

Overview

- History
- Assignment and Name conventions
- Basic Data types
- Sequences types: Lists, Tuples, and Strings
- Mutability
- Modules
- Errors and Exceptions

Brief History of Python

- Invented in the Netherlands, early 90s by Guido van Rossum
- Open source
- It is much more than a scripting language,
- Interpreted, object oriented and functional language

Running Python

The Python Interpreter

- Download from http://python.org/download/
- Typical Python implementations offer both an interpreter and compiler

On Ubuntu :

[mooc-34@edx ~]$ ***python***
Python 2.7.3 (default, Feb 27 2014, 19:58:35)
Type "help", "copyright", "credits" or "license" for more information.
>>>

On Unix:

% python
>>> 3+3
6

- Python prompts with '>>>'.

The Python Interpreter

- Download from http://python.org/download/
- Typical Python implementations offer both an interpreter and compiler

On Ubuntu :

[mooc-34@edx ~]$ ***python***
Python 2.7.3 (default, Feb 27 2014, 19:58:35)
Type "help", "copyright", "credits" or "license" for more information.
>>>

On Unix:

% python
>>> 3+3
6

- Python prompts with '>>>'.

Running Interactively on UNIX

- To exit Python
  - *In Unix, type CONTROL-D*
  - *In Windows, type CONTROL-Z + <Enter>*

Example :

```
>>> def cube(x):
...   return x * x * x
...
>>> map(cube, [1, 2, 3, 4])
[1, 8, 27, 64]
>>>
```

Running Programs on Ubuntu

- Extension : “.py”
- Calling python program via the python interpreter

```
>>> python intern.py
```

- Make a python file (For eg. `intern.py`) directly executable by
  - Include the following line as the first line of python file

```
#!$/usr/bin/env python
```

  - For making the file executable run the following command,
  - `$ sudo chmod +x intern.py`
  - Invoking file from Unix command line
  - `$ ./intern.py`

The Basics

Basic Datatypes

- Are Immutable
- 4 types :
  - Long
  - Int
  - complex (complex numbers)
  - Floats x = 3.456
- Strings
  - Can use “” or ‘’ to specify with “abc” == ‘abc’
  - Unmatched can occur within the string: “m att’s”
  - Use triple double-quotes for multi-line strings or strings than contain both ‘ and “ inside of them:
  - “““a‘b“c”””

Whitespace

Whitespace is meaningful in Python: especially indentation and placement of newlines

- Use a newline to end a line of code
  - Use `\` when must go to next line prematurely
- No braces `{}` to mark blocks of code, use *consistent* indentation instead
  - First line with *less* indentation is outside of the block
  - First line with *more* indentation starts a nested block
- Colons start of a new block in many constructs, e.g. function definitions, if then clauses, for loops etc.

Comments

- Start comments with #, rest of line is ignored
- Can include a “documentation string” as the first line of a new function or class you define
- Development environments, debugger, and other tools use it: it’s good style to include one

```
def fact(n):
  """fact(n) assumes n is a positive integer
and returns facorial of n."""
assert(n>0)
   return 1 if n==1 else n*fact(n-1)
```

Assignment

- *Binding a variable* in Python means setting a *name* to hold a *reference* to some *object*
  - *Assignment creates references, not copies*
- Names in Python do not have an intrinsic type, objects have types
  - Python determines the type of the reference automatically based on what data is assigned to it
- You create a name the first time it appears on the left side of an assignment expression:
- `x = 3`
- A reference is deleted via garbage collection after any names bound to it have passed out of scope

Naming Rules

- Names are case sensitive and cannot start with a number. They can contain letters, numbers, and underscores.

```
bob   Bob   _bob   _2_bob_   bob_2   BoB
```

- There are some reserved words:

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

Assignment

- You can assign to multiple names at the same time

```
>>> x, y = 2, 3
>>> x
2
>>> y
3
```

This makes it easy to swap values

```
>>> x, y = y, x
```

- Assignments can be chained

```
>>> a = b = x = 2
```

Accessing Non-Existent Name

Accessing a name before it's been properly created (by placing it on the left side of an assignment), raises an error

```
>>> y

Traceback (most recent call last):
  File "<pyshell#16>", line 1, in -toplevel-
    y
NameError: name 'y' is not defined
>>> y = 3
>>> y
3
```

Control Flow

- The While statement

```
count = 0
while (count < 9):
    print 'The count is:', count
    count = count + 1
```

- for Statements

```
words = ['cat', 'window', 'defenestrate']
for w in words:
    print w
```

Control Flow

- The if statement

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

- range() Function

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

Control Flow

- pass / break / continue Statements

```
for letter in 'Python':
    if letter == 'h':
        break
    print 'Current Letter :', letter
```

Functions

- Defining Functions:

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

Keyword Arguments

- Functions can be called using *keyword arguments* of the form kwarg=value. For instance, the following function:

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

- accepts one required argument (voltage) and three optional arguments (state, action, and type).

Function Call

- This function can be called in any of the following ways:
- *parrot(1000)* # 1 positional argument
- *parrot(voltage=1000)* # 1 keyword argument
- *parrot(voltage=1000000, action='VOOOOOM')* #2 keyword arguments
- *parrot(action='VOOOOOM', voltage=1000000)* # 2 keyword arguments
- *parrot('a million', 'bereft of life', 'jump')* # 3 positional arguments
- *parrot('a thousand', state='pushing up the daisies')* # 1 positional, 1 keyword

Java vs Python

JAVA	PYTHON
All variable names (along with their types) must be explicitly declared.	In Python, you never declare anything.
Java container objects (e.g. *Vector* and *ArrayList*) hold objects of the generic type *Object*, but cannot hold primitives such as *int*.	Python container objects (e.g. lists and dictionaries) can hold objects of any type, including numbers and lists. When you retrieve an object from a container, it remembers its type, so no casting is required.

public class HelloWorld
{
  public static void main (String[] args)
  {
  System.out.println("Hellold!");
  }
}
	print "Hello, world!"

Each top-level public class must be defined in its own file. If your application has 15 such classes, it has 15 files.	Multiple classes can be defined in a single file. If your application has 15 classes, the entire application could be stored in a single file,

Sequence types: Tuples, Lists, and Strings

Sequence Types

1. Tuple: (‘john’, 32, [CMSC])
   - A simple *immutable* ordered sequence of items
   - Items can be of mixed types, including collection types
2. Strings: “John Smith”
   - *Immutable*
   - Conceptually very much like a tuple
3. List: [1, 2, ‘john’, (‘up’, ‘down’)]
   - *Mutable* ordered sequence of items of mixed types

Similar Syntax

- All three sequence types (tuples, strings, and lists) share much of the same syntax and functionality.
- Key difference:
  - Tuples and strings are *immutable*
  - Lists are *mutable*
- The operations shown in this section can be applied to *all* sequence types
  - most examples will just show the operation performed on one

Sequence Types 1

- Define tuples using parentheses and commas

```
>>> tu = (23, ‘abc’, 4.56, (2,3), ‘def’)
```

- Define lists are using square brackets and commas

```
>>> li = [“abc”, 34, 4.34, 23]
```

- Define strings using quotes (“, ‘, or “““).

```
>>> st = “Hello World”
>>> st = ‘Hello World’
>>> st = “““This is a multi-line
string that uses triple quotes.”””
```

Sequence Types 2

- Access individual members of a tuple, list, or string using square bracket "array" notation
- *Note that all are 0 based…*

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
>>> tu[1]     # Second item in the tuple.
 'abc'

>>> li = ["abc", 34, 4.34, 23]
>>> li[1]      # Second item in the list.
 34

>>> st = "Hello World"
>>> st[1]   # Second character in string.
 'e'
```

Positive and negative indices

```
>>> t = (23, ‘abc’, 4.56, (2,3),
‘def’)
```

Positive index: count from the left, starting with 0

```
 >>> t[1]
‘abc’
```

Negative index: count from right, starting with –1

```
>>> t[-3]
4.56
```

Slicing: return copy of a subset

```
>>> t = (23, ‘abc’, 4.56, (2,3),
‘def’)
```

Return a copy of the container with a subset of the original members. Start copying at the first index, and stop copying *before* second.

```
>>> t[1:4]
(‘abc’, 4.56, (2,3))
```

Negative indices count from end

```
>>> t[1:-1]
(‘abc’, 4.56, (2,3))
```

Slicing: return copy of a =subset

```
>>> t = (23, ‘abc’, 4.56, (2,3),
‘def’)
```

Omit first index to make copy starting from beginning of the container

```
>>> t[:2]
(23, ‘abc’)
```

Omit second index to make copy starting at first index and going to end

```
>>> t[2:]
(4.56, (2,3), ‘def’)
```

Copying the Whole Sequence

- [:] makes a *copy* of an entire sequence

```
>>> t[:]
(23, ‘abc’, 4.56, (2,3), ‘def’)
```

- Note the difference between these two lines for mutable sequences

```
>>> l2 = l1 # Both refer to 1 ref,
            # changing one affects both
>>> l2 = l1[:] # Independent copies,
two refs
```

The ‘in’ Operator

- Boolean test whether a value is inside a container:

```
>>> t = [1, 2, 4, 5]
>>> 3 in t
False
>>> 4 in t
True
>>> 4 not in t
False
```

- For strings, tests for substrings

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

The + Operator

The + operator produces a *new* tuple, list, or string whose value is the concatenation of its arguments.

```
>>> (1, 2, 3) + (4, 5, 6)
 (1, 2, 3, 4, 5, 6)

>>> [1, 2, 3] + [4, 5, 6]
 [1, 2, 3, 4, 5, 6]

>>> “Hello” + “ ” + “World”
 ‘Hello World’
```

The * Operator

- The * operator produces a *new* tuple, list, or string that “repeats” the original content.

```
>>> (1, 2, 3) * 3
(1, 2, 3, 1, 2, 3, 1, 2, 3)

>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]

>>> “Hello” * 3
‘HelloHelloHello’
```

Mutability: Tuples vs. Lists

Lists are mutable

```
>>> li = ['abc', 23, 4.34, 23]
>>> li[1] = 45
>>> li
['abc', 45, 4.34, 23]
```

- We can change lists *in place.*
- Name *li* still points to the same memory reference when we're done.

Tuples are immutable

```
>>> t = (23, ‘abc’, 4.56, (2,3), ‘def’)
>>> t[2] = 3.14

Traceback (most recent call last):
  File "<pyshell#75>", line 1, in -toplevel-
    tu[2] = 3.14
TypeError: object doesn't support item assignment
```

- You can’t change a tuple.
- You can make a fresh tuple and assign its reference to a previously used name.

```
>>> t = (23, ‘abc’, 3.14, (2,3), ‘def’)
```

- *The immutability of tuples means they’re faster than lists.*

Operations on Lists Only

```
>>> li = [1, 11, 3, 4, 5]

>>> li.append('a')  # Note the method
syntax
>>> li
[1, 11, 3, 4, 5, 'a']

>>> li.insert(2, 'i')
>>>li
[1, 11, 'i', 3, 4, 5, 'a']
```

The *extend* method vs +

- *+* creates a fresh list with a new memory ref
- *extend* operates on list `li` in place.

```
>>> li.extend([9, 8, 7])
>>> li
[1, 2, ‘i’, 3, 4, 5, ‘a’, 9, 8, 7]
```

- *Potentially confusing*:
  - *extend* takes a list as an argument.
  - *append* takes a singleton as an argument.

```
>>> li.append([10, 11, 12])
>>> li
[1, 2, ‘i’, 3, 4, 5, ‘a’, 9, 8, 7, [10, 11,
12]]
```

Operations on Lists Only

Lists have many methods, including index, count, remove, reverse, sort

```
>>> li = ['a', 'b', 'c', 'b']
>>> li.index('b')  # index of 1st occurrence
1
>>> li.count('b')  # number of occurrences
2
>>> li.remove('b') # remove 1st occurrence
>>> li
  ['a', 'c', 'b']
```

Operations on Lists Only

```
>>> li = [5, 2, 6, 8]

>>> li.reverse()    # reverse the list *in place*
>>> li
  [8, 6, 2, 5]

>>> li.sort()       # sort the list *in place*
>>> li
  [2, 5, 6, 8]

>>> li.sort(some_function)
    # sort in place using user-defined comparison
```

Tuple details

- The comma is the tuple creation operator, not parentheses

```
>>> 1,
(1,)
```

- Python shows parentheses for clarity

```
>>> (1,)
(1,)
```

- Don't forget the comma!

```
>>> (1)
1
```

- Trailing comma only required for singletons
- Empty tuples have a special syntactic form

```
>>> ()
()
>>> tuple()
()
```

Summary: Tuples vs. Lists

- Lists slower but more powerful than tuples
  - Lists can be modified, and they have lots of handy operations and mehtods
  - Tuples are immutable and have fewer features
- To convert between tuples and lists use the list() and tuple() functions:

```
li = list(tu)
tu = tuple(li)
```

MODULES

- What is a Module?

A module is a file containing Python definitions and statements. The file name is the module name with the suffix .py appended. Within a module, the module’s name (as a string) is available as the value of the global variable __name__. For instance, use your favorite text editor to create a file called fibo.py in the current directory with the following contents

fibo.py

```
# Fibonacci numbers module
  def fib(n): # write Fibonacci series up to n
      a, b = 0, 1
      while b < n:
        print b,
        a, b = b, a+b
      print “”
 def fib2(n): # return Fibonacci series up to n
      result = []
      a, b = 0, 1
      while b < n:
         result.append(b)
         a, b = b, a+b
      return result
```

MODULE

Now enter the Python interpreter and import this module with the following command:

```
>>> import fibo
```

This does not enter the names of the functions defined in fibo directly in the current symbol table; it only enters the module name fibo there. Using the module name you can access the functions:

```
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
 [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

The Module Search Path

When a module named createlist.py is imported, the interpreter first searches for a built-in module with that name. If not found, it then searches for a file named createlist.py in a list of directories given by the variable sys.path.

sys.path is initialized from these locations:

- The directory containing the input script (or the current directory when no file is specified).
- PYTHONPATH (a list of directory names, with the same syntax as the shell variable PATH).

Errors and Exception

- Syntax Errors

Syntax errors (or parsing errors), are perhaps the most common kind of complaint you get while you are still learning Python:

```
>>> while True print 'Hello world‘
      File "<stdin>", line 1, in ?
      while  True   print 'Hello world’
                                ^
     SyntaxError:  invalid syntax
```

The parser repeats the offending line and displays a little ‘arrow’ pointing at the earliest point in the line where the error was detected. The error is caused by (or at least detected at) the token *preceding* the arrow: in the example, the error is detected at the function print(), since a colon (':') is missing before it. File name and line number are printed so you know where to look in case the input came from a script.

Errors and Exception

- Exceptions:

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called *exceptions* . Most exceptions are not handled by programs, however, and result in error messages as shown here:

```
>>> 10 * (1/0)
 Traceback (most recent call last): File "<stdin>", line 1, in ?
ZeroDivisionError: division by zero
 >>> 4 + spam*3
Traceback (most recent call last): File "<stdin>", line 1, in ?
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last): File "<stdin>", line 1, in ?
TypeError: Can't convert 'int' object to str implicitly
```

Handling Exception

It is possible to write programs that handle selected exceptions. Look at the following example, which asks the user for input until a valid integer has been entered, but allows the user to interrupt the program (using Control-C or whatever the operating system supports); note that a user-generated interruption is signalled by raising the KeyboardInterrupt exception.

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

Handling Exception

The try statement works as follows.

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

Enough to Understand the Code

- Indentation matters to code meaning
  - Block structure indicated by indentation
- First assignment to a variable creates it
  - Variable types don't need to be declared.
  - Python figures out the variable types on its own.
- Assignment is `=` and comparison is `==`
- For numbers `+ - * / %` are as expected
  - Special use of `+` for string concatenation and `%` for string formatting (as in C's printf)
- Logical operators are words (`and, or, not`) *not* symbols
- The basic printing command is `print`

End

….Thank You