Python Programming

By

Subhadip Mukherjee

Dept. of Computer Science and BCA,
Kharagpur College,
Kharagpur, India

LECTURE 1

For CBCS 4th Semester Students

ABOUT PYTHON

- Development started in the 1980's by Guido van Rossum.
  - Only became popular in the last decade or so.
- Python 2.x currently dominates, but Python 3.x is the future of Python.
- Interpreted, very-high-level programming language.
- Supports a multitude of programming paradigms.
  - OOP, functional, procedural, logic, structured, etc.
- General purpose.
  - Very comprehensive standard library includes numeric modules, crypto services, OS interfaces, networking modules, GUI support, development tools, etc.

PHILOSOPHY

From *The Zen of Python* (https://www.python.org/dev/peps/pep-0020/)

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
Namespaces are one honking great idea -- let's do more of those!

NOTABLE FEATURES

- Easy to learn.
- Supports quick development.
- Cross-platform.
- Open Source.
- Extensible.
- Embeddable.
- Large standard library and active community.
- Useful for a wide variety of applications.

GETTING STARTED

Before we can begin, we need to actually install Python!

It is recommended that you set up a Linux virtual machine – this will save you a lot of headache later on in the course.

You can use any VM software you'd like, but I recommend Virtual Box. To ensure that the class is all on the same page, I'd also recommend using Ubuntu 14.04 as your virtualized OS. This will make installing and setting up packages much easier for you.

Your very first task is to set up your VM, make sure Python 2.7 is installed (it should be!) and write a simple Hello World program to make sure you're good to go. Do not put this off until Friday (when your first assignment is due)!

GETTING STARTED

- Choose and install an editor.
  - For Linux, I prefer SublimeText.
  - Windows users will likely use Idle by default.
  - Options include vim, emacs, Notepad++, PyCharm, Eclipse, etc.

Throughout this course, I will be using SublimeText in an Ubuntu environment for all of the demos.

INTERPRETER

- The standard implementation of Python is interpreted.
  - You can find info on various implementations here.
- The interpreter translates Python code into bytecode, and this bytecode is executed by the Python VM (similar to Java).
- Two modes: normal and interactive.
  - Normal mode: entire .py files are provided to the interpreter.
  - Interactive mode: read-eval-print loop (REPL) executes statements piecewise.

INTERPRETER: NORMAL MODE

Let's write our first Python program!

In our favorite editor, let's create helloworld.py with the following contents:

```
print "Hello, World!"
```

From the terminal:

```
$ python helloworld.py
Hello, World!
```

Note: In Python 2.x, print is a statement. In Python 3.x, it is a function. If you want to get in the 3.x habit, include at the beginning:

```
from __future__ import print_function
```

Now, you can write

```
print("Hello, World!")
```

INTERPRETER: NORMAL MODE

Let's include a she-bang in the beginning of helloworld.py:

```
#!/usr/bin/env python
print "Hello, World!"
```

Now, from the terminal:

```
$ ./helloworld.py
Hello, World!
```

INTERPRETER: INTERACTIVE MODE

Let's accomplish the same task (and more) in interactive mode.

Some options:
-c : executes single command.
-O: use basic optimizations.
-d: debugging info.
More can be found here.

```
$ python
>>> print "Hello, World!"
Hello, World!
>>> hellostring = "Hello, World!"
>>> hellostring
'Hello, World!'
>>> 2*5
10
>>> 2*hellostring
'Hello, World!Hello, World!'
>>> for i in range(0,3):
...     print "Hello, World!"
...
Hello, World!
Hello, World!
Hello, World!
>>> exit()
$
```

SOME FUNDAMENTALS

- Whitespace is significant in Python. Where other languages may use {} or (), Python uses indentation to denote code blocks.

- Comments
  - Single-line comments denoted by #.
  - Multi-line comments begin and end with three “s.
  - Typically, multi-line comments are meant for documentation.
  - Comments should express information that cannot be expressed in code – do not restate code.

```
# here’s a comment
for i in range(0,3):
    print i
def myfunc():
    """here’s a comment about
    the myfunc function"""
    print "I'm in a function!"
```

PYTHON TYPING

- Python is a strongly, dynamically typed language.
- Strong Typing
  - Obviously, Python isn't performing static type checking, but it does prevent mixing operations between mismatched types.
  - Explicit conversions are required in order to mix types.
  - Example: 2 + "four" ← not going to fly
- Dynamic Typing
  - All type checking is done at runtime.
  - No need to declare a variable or give it a type before use.

Let's start by looking at Python's built-in data types.

NUMERIC TYPES

The subtypes are int, long, float and complex.

- Their respective constructors are int(), long(), float(), and complex().

- All numeric types, except complex, support the typical numeric operations you'd expect to find (a list is available here).
- Mixed arithmetic is supported, with the "narrower" type widened to that of the other. The same rule is used for mixed comparisons.

NUMERIC TYPES

- Numeric
  - **int**: equivalent to C's long int in 2.x but unlimited in 3.x.
  - **float**: equivalent to C's doubles.
  - **long**: unlimited in 2.x and unavailable in 3.x.
  - **complex**: complex numbers.

- Supported operations include constructors (i.e. int(3)), arithmetic, negation, modulus, absolute value, exponentiation, etc.

```
$ python
>>> 3 + 2
5
>>> 18 % 5
3
>>> abs(-7)
7
>>> float(9)
9.0
>>> int(5.3)
5
>>> complex(1,2)
(1+2j)
>>> 2 ** 8
256
```

SEQUENCE DATA TYPES

There are seven sequence subtypes: strings, Unicode strings, lists, tuples, bytearrays, buffers, and xrange objects.

All data types support arrays of object but with varying limitations.

The most commonly used sequence data types are strings, lists, and tuples. The xrange data type finds common use in the construction of enumeration-controlled loops. The others are used less commonly.

SEQUENCE TYPES: STRINGS

Created by simply enclosing characters in either single- or double-quotes.

It's enough to simply assign the string to a variable.

Strings are immutable.

There are a tremendous amount of built-in string methods (listed here).

```
mystring = "Hi, I'm a string!"
```

SEQUENCE TYPES: STRINGS

Python supports a number of escape sequences such as '\t', '\n', etc.

Placing 'r' before a string will yield its raw value.

There is a string formatting operator '%' similar to C. A list of string formatting symbols is available here.

Two string literals beside one another are automatically concatenated together.

```
print "\tHello,\n"
print r"\tWorld!\n"
print "Python is " "so cool."
```

```
$ python ex.py
        Hello,

\tWorld!\n
Python is so cool.
```

SEQUENCE TYPES: UNICODE STRINGS

Unicode strings can be used to store and manipulate Unicode data.

As simple as creating a normal string (just put a ‘u’ on it!).

Use Unicode-Escape encoding for special characters.

Also has a raw mode, use ‘ur’ as a prefix.

To translate to a regular string, use the .encode() method.

To translate from a regular string to Unicode, use the unicode() method.

```
myunicodestr1 = u"Hi Class!"
myunicodestr2 = u"Hi\u0020Class!"
print myunicodestr1, myunicodestr2
newunicode = u'\xe4\xf6\xfc'
print newunicode
newstr = newunicode.encode('utf-8')
print newstr
print unicode(newstr, 'utf-8')
```

Output:

Hi Class! Hi Class!

äöü

äöü

äöü

SEQUENCE TYPES: LISTS

Lists are an incredibly useful *compound* data type.

Lists can be initialized by the constructor, or with a bracket structure containing 0 or more elements.

Lists are mutable – it is possible to change their content. They contain the additional mutable operations previously listed.

Lists are nestable. Feel free to create lists of lists of lists…

```
mylist = [42, 'apple', u'unicode apple', 5234656]
print mylist
mylist[2] = 'banana'
print mylist
mylist [3] = [['item1', 'item2'], ['item3', 'item4']]
print mylist
mylist.sort()
print mylist
print mylist.pop()
mynewlist = [x for x in range(0,5)]
print mynewlist
```

```
[42, 'apple', u'unicode apple', 5234656]
[42, 'apple', 'banana', 5234656]
[42, 'apple', 'banana', [['item1', 'item2'], ['item3', 'item4']]]
[42, [['item1', 'item2'], ['item3', 'item4']], 'apple', 'banana']
banana
[0, 1, 2, 3, 4]
```

SEQUENCE DATA TYPES

- Sequence
  - **str**: string, represented as a sequence of 8-bit characters in Python 2.x.
  - **unicode**: stores an abstract sequence of code points.
  - **list**: a compound, mutable data type that can hold items of varying types.
  - **tuple**: a compound, immutable data type that can hold items of varying types. Comma separated items surrounded by parentheses.
  - a few more – we'll cover them later.

```
$ python
>>> mylist = ["spam", "eggs", "toast"] # List of strings!
>>> "eggs" in mylist
True
>>> len(mylist)
3
>>> mynewlist = ["coffee", "tea"]
>>> mylist + mynewlist
['spam', 'eggs', 'toast', 'coffee', 'tea']
>>> mytuple = tuple(mynewlist)
>>> mytuple
('coffee', 'tea')
>>> mytuple.index("tea")
1
>>> mylonglist = ['spam', 'eggs', 'toast', 'coffee', 'tea', 'banana']
>>> mylonglist[2:4]
['toast', 'coffee']
```

Thank You