



# Machine learning

Lesson 02. Basics of python

Kirill Svyatov

Ulyanovsk State Technical University,  
Faculty of Information Systems and Technologies



# Python in general

- What is python?
  - High level programming language
  - Emphasize on code readability
  - Very clear syntax + large and comprehensive standard library
  - Use of indentation for block delimiters
  - Multiprogramming paradigm: OO, imperative, functional, procedural, reflective
  - A fully [dynamic type](#) system and automatic [memory management](#)
  - Scripting language + standalone executable program + interpreter
  - Can run on many platform: Windows, Linux, Macintosh
- Updates:
  - Newest version: 3.2.2 (CPython, JPython, IronPython)
  - Website: [www.python.org](http://www.python.org)



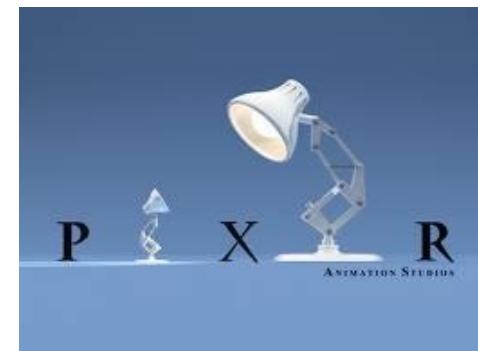
# Python in general (Cont')

- Advantages:
  - Software quality
  - Developer productivity
  - Program portability
  - Support libraries
  - Component integration
  - Enjoyment
- Disadvantages:
  - not always be as fast as that of compiled languages such as C and C++



# Python in general (Cont')

- Applications of python:



# Python in general (Cont')

- Python's Capability:
  - System Programming
  - GUI
  - Internet Scripting
  - Component Integration
  - Database Programming
  - Rapid Prototyping
  - Numeric and Scientific Programming
  - Gaming, Images, Serial Ports, XML, Robots, and More



# How Python program runs?

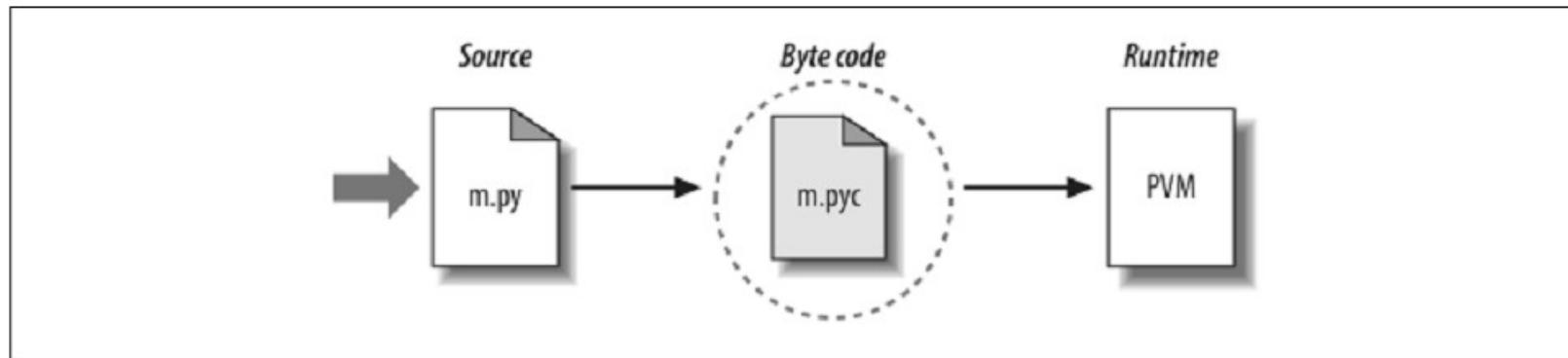


Figure 2-2. Python's traditional runtime execution model: source code you type is translated to byte code, which is then run by the Python Virtual Machine. Your code is automatically compiled, but then it is interpreted.

Notice: pure Python code runs at speeds somewhere between those of a traditional compiled language and a traditional interpreted language

HELLO WORLD

```
print "hello world"
```



## FROM INTERPRETER

```
$ python  
->>> print "hello world"  
hello world
```



# REPL

Read, Eval, Print, Loop



# REPL

```
$ python
>>> 2 + 2 # read, eval
4          # print
>>>       # repeat (loop)
```



## REPL (2)

Many developers keep a REPL handy  
during programming



## FROM SCRIPT

Make file `hello.py` with  
`print "hello world"`

Run with:

`python hello.py`



## (UNIX) SCRIPT

Make file `hello` with

```
#!/usr/bin/env python  
print "hello world"
```

Run with:

```
chmod +x hello  
../hello
```



## PYTHON 3 HELLO WORLD

`print` is no longer a statement, but a function

```
print("hello world")
```



# Objects



# OBJECTS

Everything in *Python* is an object that has:

- an *identity* (`id`)
- a *value* (mutable or immutable)



id

```
>>> a = 4  
>>> id(a)  
6406896
```



# VALUE

- **Mutable:** When you alter the item, the id is still the same. Dictionary, List
- **Immutable:** String, Integer, Tuple



## MUTABLE

```
>>> b = []
>>> id(b)
140675605442000
>>> b.append(3)
>>> b
[3]
>>> id(b)
140675605442000    # SAME!
```



# IMMUTABLE

```
>>> a = 4
>>> id(a)
6406896
>>> a = a + 1
>>> id(a)
6406872 # DIFFERENT!
```



## VARIABLES

```
a = 4          # Integer
b = 5.6        # Float
c = "hello"    # String
a = "4"        # rebound to String
```



# NAMING

- lowercase
- underscore\_between\_words
- don't start with numbers

See PEP 8



# Math



# MATH

`+, -, *, /, ** (power), % (modulo)`



## CAREFUL WITH INTEGER DIVISION

```
>>> 3/4
```

```
0
```

```
>>> 3/4.
```

```
0.75
```

(In Python 3 // is integer division operator)



What happens when you  
raise 10 to the 100th?



LONG

**>>> 10\*\*\*100**

100  
000  
000  
00000L



## *LONG (2)*

```
>>> import sys  
>>> sys.maxint  
9223372036854775807  
>>> sys.maxint + 1  
9223372036854775808L
```



# Strings



## *STRINGS*

```
name = 'matt'  
with_quote = "I ain't gonna"  
longer = """This string has  
multiple lines  
in it"""
```



# HOW DO I PRINT?

He said, “I’m sorry”



# STRING ESCAPING

Escape with \

```
>>> print 'He said, "I\'m sorry"'  
He said, "I'm sorry"  
>>> print '''He said, "I'm sorry'''  
He said, "I'm sorry"  
>>> print """He said, "I'm sorry\"""  
He said, "I'm sorry"
```



## STRING ESCAPING (2)

Escape Sequence	Output
\\	Backslash
\'	Single quote
\"	Double quote
\b	ASCII Backspace
\n	Newline
\t	Tab
\u12af	Unicode 16 bit
\U12af89bc	Unicode 32 bit
\o84	Octal character
\xFF	Hex character



# STRING FORMATTING

c-like

```
>>> "%s %s" %('hello', 'world')
'hello world'
```

PEP 3101 style

```
>>> "{0} {1}".format('hello', 'world')
'hello world'
```



# Methods & dir



# dir

Lists attributes and methods:

```
>>> dir("a string")
['__add__', '__class__', ... 'startswith', 'strip',
'swapcase', 'title', 'translate', 'upper', 'zfill']
```



Whats with all the  
'blah'?



## DUNDER METHODS

*dunder* (double under) or "special/magic" methods determine what will happen when + (`__add__`) or / (`__div__`) is called.



# help

```
>>> help("a string".startswith)
```

Help on built-in function startswith:

```
startswith(...)  
S.startswith(prefix[, start[, end]]) -> bool
```

Return True if S starts with the specified prefix, False otherwise.

With optional start, test S beginning at that position.

With optional end, stop comparing S at that position.

prefix can also be a tuple of strings to try.



## STRING METHODS

- **s.endswith(sub)**

Returns True if endswith sub

- **s.find(sub)**

Returns index of sub or -1

- **s.format(\*args)**

Places args in string



## STRING METHODS (2)

- **s.index(sub)**

Returns index of `sub` or exception

- **s.join(list)**

Returns `list` items separated by string

- **s.strip()**

Removes whitespace from start/end



# Comments



## COMMENTS

Comments follow a #



## COMMENTS

No multi-line comments



# More Types



# None

Pythonic way of saying NULL. Evaluates to False.

c = **None**



## *BOOLEANS*

a = **True**

b = **False**



## *SEQUENCES*

- *lists*
- *tuples*
- *sets*



## *LISTS*

Hold sequences.

How would we find out the attributes & methods of a list?



## *LISTS*

```
>>> dir([])  
['__add__', '__class__', '__contains__', ...  
'__iter__', ..., '__len__', ..., 'append', 'count',  
'extend', 'index', 'insert', 'pop', 'remove',  
'reverse', 'sort']
```



## LISTS

```
>>> a = []
>>> a.append(4)
>>> a.append('hello')
>>> a.append(1)
>>> a.sort() # in place
>>> print a
[1, 4, 'hello']
```



## *LISTS*

How would we find out documentation  
for a method?



## *LISTS*

help function:

```
>>> help([].append)
Help on built-in function append:

append(...)
L.append(object) -- append object to end
```



## LIST METHODS

- **l.append(x)**

Insert x at end of list

- **l.extend(l2)**

Add l2 items to list

- **l.sort()**

In place sort



## LIST METHODS (2)

- **l.reverse()**

Reverse list in place

- **l.remove(item)**

Remove first item found

- **l.pop()**

Remove/return item at end of list



# Dictionaries



## *DICTIONARIES*

Also called *hashmap* or *associative array* elsewhere

```
>>> age = {}  
>>> age['george'] = 10  
>>> age['fred'] = 12  
>>> age['henry'] = 10  
>>> print age['george']  
10
```



## *DICTIONARIES (2)*

Find out if 'matt' in age

```
>>> 'matt' in age
```

```
False
```



## in STATEMENT

Uses `__contains__` dunder method to determine membership. (Or `__iter__` as fallback)



## .get

```
>>> print age['charles']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'charles'
>>> print age.get('charles', 'Not found')
Not found
```



## DELETING KEYS

Removing 'charles' from age

```
>>> del age['charles']
```



## DELETING KEYS

`del` not in `dir.`. `pop` is an alternative



# Functions



## FUNCTIONS

```
def add_2(num):
    """return 2
    more than num
    """
    return num + 2

five = add_2(3)
```



## FUNCTIONS (2)

- `def`
- function name
- (parameters)
- `:` + indent
- optional documentation
- body
- return



## WHITE SPACE

Instead of { use a : and indent  
consistently (4 spaces)



## WHITESPACE (2)

invoke `python -tt` to error out during  
inconsistent tab/space usage in a file



## DEFAULT (NAMED) PARAMETERS

```
def add_n(num, n=3):
    """default to
    adding 3"""
    return num + n
```

```
five = add_n(2)
ten = add_n(15, -5)
```



## \_\_doc\_\_

Functions have *docstrings*. Accessible via `.__doc__` or `help`



## \_\_doc\_\_

```
>>> def echo(txt):
...     "echo back txt"
...     return txt
>>> help(echo)
Help on function echo in module __main__:
<BLANKLINE>
echo(txt)
    echo back txt
<BLANKLINE>
```



## NAMING

- lowercase
- underscore\_between\_words
- don't start with numbers
- verb

See PEP 8



# Conditionals



## CONDITIONALS

```
if grade > 90:  
    print "A"  
elif grade > 80:  
    print "B"  
elif grade > 70:  
    print "C"  
else:  
    print "D"
```



Remember the  
colon/whitespace!



## BOOLEANS

a = **True**

b = **False**



## COMPARISON OPERATORS

Supports (`>`, `>=`, `<`, `<=`, `==`, `!=`)

```
>>> 5 > 9
```

```
False
```

```
>>> 'matt' != 'fred'
```

```
True
```

```
>>> isinstance('matt',  
               basestring)
```

```
True
```



## BOOLEAN OPERATORS

and, or, not (for logical), &, |, and ^ (for bitwise)

```
>>> x = 5
```

```
>>> x < -4 or x > 4
```

```
True
```



## BOOLEAN NOTE

Parens are only required for precedence

```
if (x > 10):  
    print "Big"
```

same as

```
if x > 10:  
    print "Big"
```



## CHAINED COMPARISONS

```
if 3 < x < 5:  
    print "Four!"
```

Same as

```
if x > 3 and x < 5:  
    print "Four!"
```



# Iteration



## ITERATION

```
for number in [1,2,3,4,5,6]:  
    print number
```

```
for number in range(1, 7):  
    print number
```



## range NOTE

Python tends to follow *half-open interval* (`[start, end)`) with `range` and slices.

- $\text{end} - \text{start} = \text{length}$
- easy to concat ranges w/o overlap



## ITERATION (2)

Java/C-esque style of object in array  
access (BAD):

```
animals = ["cat", "dog", "bird"]
for index in range(len(animals)):
    print index, animals[index]
```



## ITERATION (3)

If you need indices, use `enumerate`

```
animals = ["cat", "dog", "bird"]
for index, value in enumerate(animals):
    print index, value
```



## ITERATION (4)

Can break out of nearest loop

```
for item in sequence:  
    # process until first negative  
    if item < 0:  
        break  
    # process item
```



## ITERATION (5)

Can continue to skip over items

```
for item in sequence:  
    if item < 0:  
        continue  
    # process all positive items
```



## ITERATION (6)

Can loop over lists, strings, iterators, dictionaries... sequence like things:

```
my_dict = { "name": "matt", "cash": 5.45}
for key in my_dict.keys():
    # process key

for value in my_dict.values():
    # process value

for key, value in my_dict.items():
    # process items
```



pass

pass is a null operation

```
for i in range(10):
    # do nothing 10 times
    pass
```



## HINT

Don't modify *list* or *dictionary* contents  
while looping over them



# Slicing



## SLICING

Sequences (lists, tuples, strings, etc) can  
be *sliced* to pull out a single item

```
my_pets = ["dog", "cat", "bird"]
favorite = my_pets[0]
bird = my_pets[-1]
```



## NEGATIVE INDEXING

Proper way to think of [negative indexing] is to reinterpret  $a[-X]$  as  $a[\text{len}(a)-X]$

@gvanrossum



## SLICING (2)

Slices can take an end index, to pull out a list of items

```
my_pets = ["dog", "cat", "bird"]
# a list
cat_and_dog = my_pets[0:2]
cat_and_dog2 = my_pets[:2]
cat_and_bird = my_pets[1:3]
cat_and_bird2 = my_pets[1:]
```



## SLICING (3)

Slices can take a stride

```
my_pets = ["dog", "cat", "bird"]
# a list
dog_and_bird = [0:3:2]
zero_three_etc = range(0,10)
[::3]
```



## SLICING (4)

Just to beat it in

```
veg = "tomatoe"  
correct = veg[:-1]  
tmte = veg[::2]  
eotamot = veg[::-1]
```



# File IO



## FILE INPUT

Open a file to read from it (old style):

```
fin = open("foo.txt")
for line in fin:
    # manipulate line
fin.close()
```



## FILE OUTPUT

Open a file using '`w`' to write to a file:

```
fout = open("bar.txt", "w")
fout.write("hello world")
fout.close()
```



Always remember to  
close your files!



## CLOSING WITH `with`

implicit close (new 2.5+ style)

```
with open('bar.txt') as fin:  
    for line in fin:  
        # process line
```



# Classes



# CLASSES

```
class Animal(object):
    def __init__(self, name):
        self.name = name

    def talk(self):
        print "Generic Animal Sound"

animal = Animal("thing")
animal.talk()
```



## CLASSES (2)

notes:

- `object` (base class) (fixed in 3.X)
- `dunder init` (constructor)
- all methods take `self` as first parameter



## CLASSES(2)

### Subclassing

```
class Cat(Animal):
    def talk(self):
        print '%s says, "Meow!"' % (self.name)

cat = Cat("Groucho")
cat.talk() # invoke method
```



## CLASSES(3)

```
class Cheetah(Cat):
    """classes can have
    docstrings"""
```

```
def talk(self):
    print "Growl"
```



## NAMING

- CamelCase
- don't start with numbers
- Nouns



# Debugging



## Poor Mans

`print` works a lot of the time



## REMEMBER

Clean up `print` statements. If you really need them, use `logging` or write to `sys.stdout`



**pdb**

```
import pdb; pdb.set_trace()
```



## **pdb COMMANDS**

- **h** - help
- **s** - step into
- **n** - next
- **c** - continue
- **w** - where am I (in stack)?
- **l** - list code around me

