Python Tutorial

Klaus Potzmader

Graz University of Technology

March 26, 2012

Slides based on previous work by Jan Pöschko
Why Python?

Python is:

- very readable
- easy to learn
- interpreted & interactive – like a UNIX shell, only better
- object-oriented – but not religious about it
- slower than C, but it is easy to integrate C, Fortran, Java

“Batteries included”
The Zen of Python

• 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.
• If the implementation is hard to explain, it’s a bad idea.
• If the implementation is easy to explain, it may be a good idea.
• Namespaces are one honking great idea – let’s do more of those!
Installing Python

- included in most distributions of Linux and Mac OS X
- downloadable from http://www.python.org/download/
- libraries:
  - NumPy: http://sourceforge.net/projects/numpy/
  - matplotlib: http://sourceforge.net/projects/matplotlib/
  - NetworkX: http://networkx.lanl.gov/install.html
- editors:
  - Eclipse
  - emacs, vim
  - ...(anything that knows how to handle tabs and spaces)
Invoking Python

- Interactive mode: Open a console/terminal window and run `python` (fancier alternative: `ipython`)

```
$ python2.6
Python 2.6.6 (r266:84292, Mar 25 2011, 19:24:58)
[GCC 4.5.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import this
The Zen of Python, by Tim Peters

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.
```

- Executing files: `python myfile.py`
- Make sure you have python 2.6 installed as it’s the version used in our tests
  - On some linux distributions, it’s installed next to python 2.7 and can be invoked using the command `python2.6`
Language basics

- no variable declarations
- dynamically typed

String delimiters:

```python
s = " Hello world!"
s = ' Hello world!'
s = """ Hello
world!""" # multi-line string
```
Booleans

- True is true, False is false
- 0, "" and None are false, (almost) everything else is true

```
not A
A and B
A or B
```

Comparisons: ==, !=, <, <=, >, >=

```
2 == 2
1 < 2 <= 3
1 == 2 and "3" or "4"  # = "4"
```
Lists

```python
L = [1, 2, 3]
L[0]                          # = 1
L[1:3]                        # = [2, 3]
L[:−1]                        # = [1, 2]
L.append(4)                   # → [1, 2, 3, 4]
L += [5, 6]                   # → [1, 2, 3, 4, 5, 6]
del L[5]                      # → [1, 2, 3, 4, 5]
len(L)                        # = 5
L.reverse()                   # → [5, 4, 3, 2, 1]
L.sort()                      # → [1, 2, 3, 4, 5]
```

Variables only hold references to lists (like to everything else):

```python
M = L
```
Dictionaries:

D = {"Mozart": 1756, "Schubert": 1797}
D["Mozart"] # = 1756
D.keys() # = ["Schubert", "Mozart"]
D.values() # = [1797, 1756]
D.update({"Beethoven": 1770})
D["Einstein"] = 1879
"Einstein" in D # = True
len(D) # = 4
D.get("Newton", "unknown") # = 'unknown'

Sets:

s = set([1, "hello", "world" ])
s.add(3)
2 in s # = False
Introduction

Output and string formatting

Basic printing

```python
print "two:", 2, "four:", 4  # two: 2 four: 4
print "%0.2f + %d = %0.2f" %(2.5, 2, 2.5+2)
# 2.50 + 2 = 4.50
```

`str.format()`

```python
formatted = "PI: {0:0.2f}".format(math.pi)
print formatted  # PI: 3.14
```

Newline peculiarities:

```python
print "text"  # appends <newline>
print "text", # appends a space (note the comma)

import sys
sys.stdout.write("text")  # appends nothing
```

http://docs.python.org/library/string.html#formatstrings
Control flow

```python
if n == 0:
    print "n is 0"
elif n > 0:
    print "n is positive"
else:
    print "n is negative"
```

- Indentation matters!
- Don’t mix tabs and spaces – configure your editor appropriately!

```python
while n > 0:
    n -= 1
```

```python
for n in [1, 2, 3, 4]:
    print n
```
Iterate over a dictionary and format strings:

\[
D = \{ "Mozart": 1756, "Schubert": 1797 \}
\]

for name, year in D.items():
    print "%s was born in %d" % (name, year)

Enumerate list:

\[
L = ["item1", "item2", "item3", "item4"]
\]

for i, item in enumerate(L):
    print "The list contains %s at index %d" \ 
    % (item, i)

List comprehensions:

\[
quad = [x**2 for x in range(8)]
\]

# = [0, 1, 4, 9, 16, 25, 36, 49]

even = [x**2 for x in range(8) if x % 2 == 0]

# = [0, 4, 16, 36]
values = [[1, 0], [1, 1]]

# Join values by "," and lines by newlines
csvtext = '\n'.join(', '.join(str(value) + ' ' + for value in line) for line in values)

# Write the text into a file
csvfile = open('file.csv', 'w')
csvfile.write(csvtext)
csvfile.close()

with open("myfile.txt") as f:
    data = f.read()
    # do something
# file is closed upon leaving the 'with' block
Functions: \( \text{fac}(n) = n! := \prod_{k=1}^{n} k \)

```python
def fac(n):
    if n == 1:
        return 1
    else:
        return n * fac(n - 1)

def fac(n):
    result = 1
    for k in range(1, n + 1):
        result *= k
    return result

def fac(n):
    return reduce(lambda a, b: a * b, range(1, n + 1), 1)
```
Parameters point to values (like all variables):

```python
def change(a, b):
    a = b[0] = 0
a, b = 1, [1, 1]
change(a, b)  # → a = 1, b = [0, 1]
```

Named Parameters:

```python
def helloworld( hello="Hello", world="world" ):
    print hello, world

helloworld()  # → "Hello world"
helloworld( hello="Hi" )  # → "Hi world"
helloworld( world="earth" )  # → "Hello earth"
# position does not matter:
helloworld( world="earth", hello="Hi" )  # → "Hi earth"
```
Functions: Parameters 2

Special list and keyword (dictionary) arguments
- starred for list, double-starred for dictionary
- Enable arbitrary number of arguments
- Restriction if used in combination: keyword arguments need to be after the list arguments

```python
def fun(*args, **kwargs):
    # do something with values
fun("some", "list", "values", 
    and="key", value="pairs", as="dictionary")
```

E.g.:

```python
def sum(*values):
    result = 0
    for value in values:
        result += value
    return result
```
Iterables

- Basically everything you can use in for .. in ..
- e.g.

```python
mylist = [x**2 for x in range(2, 5)]
for i in mylist:
    print i

mystr = "helloworld"
for char in mystr:
    print char
```

- Can be read multiple times
- Downside: Everything is in memory as a whole
Generators

- Special iterables that generate values on the fly
- Not in memory as a whole
- Thus, only readable once

Same example as before, with generators:

```python
# notice the different braces
mylist = (x**2 for x in range(2, 5))
print(mylist)  # -> <generator object ...>
for i in mylist:
    print(i)

# a second loop won't print anything as the generator already went through all items
```

- as soon as the loop wants to read a new value, it is 'generated' from the comprehension rule given above
Generators 2

```python
def reverse(data):
    for index in range(len(data)−1, −1, −1):
        yield data[index]

for char in reverse('webscience'):
    print(char)
```

- Yield temporarily suspends processing and remembers the execution state
- Upon resume, the generator picks up where it left off
- More at http://docs.python.org/tutorial/classes.html#generators and/or
  http://stackoverflow.com/a/231855
Read file line by line and split by ‘,’:

```python
with open('textfile.txt', 'r') as f:
    lines = (line.rstrip('
').split(',')
             for line in f)
    for line in lines:
        print line # process here
```

E.g. if testfile.txt contains

one,two,three
3,2,1

we get

['one', 'two', 'three']
['3', '2', '1']
```python
class Animal:
    def __init__(self, name):
        self.name = name
    def say_hello(self):
        print(f"I'm {self.name}"

class Dog(Animal):
    def __init__(self, name, owner):
        super(Dog, self).__init__(name)
        self.owner = owner
    def say_hello(self):
        print(f"Hello {self.owner}"

dog = Dog("Charly", "Mike")
dog.say_hello()
```
Any Python file (e.g. `mymodule.py`) is a module and can be imported:

```python
import mymodule
mymodule.myfunction()
```

```python
from mymodule import myfunction
from mymodule import *  # rather discouraged
myfunction()
```

File-system directories can be used to create “namespaces”, if they contain a file `__init__.py` (which may contain initialization code):

```python
import mydir.mymodule
from mydir.mymodule import myfunction
```
Variable `__name__` contains the name of the current module. It equals "`__main__" when the script is run directly from the command-line.

Common idiom:

```python
def main():
    print "Hello world!"

if __name__ == "__main__":
    main()
```
More advanced concepts

• exceptions
• multiple inheritance
• operator overloading
• metaclasses
• inline documentation and test code
• extensive “runtime” information
Regular expressions

re is a module for handling regular expressions.

```python
import re

re.findall(r'[A-Za-z]+', 'Hello world!')
# = ['Hello', 'world']
```

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>0 or more</td>
<td>findall(pattern, string)</td>
</tr>
<tr>
<td>+</td>
<td>1 or more</td>
<td>match(pattern, string)</td>
</tr>
<tr>
<td>?</td>
<td>0 or 1</td>
<td>sub(pattern, repl, string)</td>
</tr>
<tr>
<td>[...]</td>
<td>certain characters</td>
<td>split(pattern, string)</td>
</tr>
<tr>
<td>[^...]</td>
<td>characters excluded</td>
<td>...</td>
</tr>
</tbody>
</table>

Pre-compiling regular expressions:

```python
WORD_RE = re.compile(r'[A-Za-z]+')
WORD_RE.findall('Hello world!')
```
urlib2 is a module for opening URLs.

```python
from urllib2 import urlopen

urlfile = urlopen("http://www.google.com")
content = urlfile.read()

# = '<!doctype html><html> [...]<script>'

Basic HTTP authentication:

```python
import urllib2

auth_handler = urllib2.HTTPBasicAuthHandler()
auth_handler.add_password(realm='the realm',
                           uri='http://www.example.com',
                           user='usr', passwd='pwd')

opener = urllib2.build_opener(auth_handler)

urllib2.install_opener(opener)

urlopen('http://www.example.com/restricted')
```
xml.dom is a module for parsing XML documents and accessing them using the Document Object Model.

```python
from xml.dom.minidom import parseString

content = """<tag attr="1">
  <sub>text</sub></tag>"""

dom = parseString(content)

print(dom.childNodes[0].getAttribute('attr'))  # = '1'

subs = dom.getElementsByTagName('sub')[0].childNodes[0].nodeValue  # = 'text'
```
JSON data

`json` is a module for parsing and exporting JSON data snippets.

```python
import json

json_data = '{"list": [1, 2], "key": "value"}'
contents = json.loads(json_data)
print contents['list']  # -> [1,2]
print contents['key']  # -> value
```
networkx is a module for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks.

```python
import networkx as nx
G = nx.Graph()
G.add_node(1)
G.add_edge(1, 2)
G.add_edge(2, 3)

Plot graphs:

import matplotlib.pyplot as plt
plt.figure()
G.draw()
plt.savefig('graph.png')
```
NetworkX: Customized visualization

```python
plt.figure()
plt.axis('off')
pos = nx.spring_layout(G, iterations=50)
nx.draw_networkx_edges(G, pos)
nx.draw_networkx_nodes(G, pos, [1], node_color='r')
nx.draw_networkx_nodes(G, pos, [2, 3], node_color='b')
nx.draw_networkx_labels(G, pos)
plt.savefig('graph.png', dpi=72)
```
Further reading

- Official Python tutorial: http://docs.python.org/tutorial/
- Python tutorial at the University of Toronto: http://www.cs.toronto.edu/~gpenn/csc401/401_python_web/
- Python 2.6 Quick Reference: http://rgruet.free.fr/PQR26/PQR2.6.html
- http://docs.python.org/library/re.html
- http://docs.python.org/library/urllib2.html
- http://docs.python.org/library/xml.dom.html
- http://docs.python.org/library/json.html
- NetworkX: http://networkx.lanl.gov/contents.html