Python in practice

Life is better without braces

Robert Wojciechowicz

Tieto Poland Sp. z o.o.

Python workshop, 2012

Try it in a Python interactive interpreter :-)

```
>>> from __future__ import braces
  File "<stdin>", line 1
SyntaxError: not a chance
```

Outline

- The short introduction
- Overview of Python concepts
  - Data model
  - Execution model
  - Modules and packages
  - Iterators
  - Generators
- Idioms
  - Good practices
  - Efficiency suggestions
- Gotchas
  - Common mistakes
  - Anti-idioms

Outline (cont.)

- Unicode
- Object oriented programming
  - Signature-based polymorphism
  - Special methods
  - Properties
- Concurrency
  - Threads
  - Processes
  - Asyncore
- Testing
  - Unittests
  - Doctests
  - Nose

Outline (cont.)

- XML
  - ElementTree API
  - lxml - ElementTree API extension
- Optimization
  - Profiling tools
  - Optimization techniques

Zen

```
>>> import this
The Zen of Python, by Tim Peters

[...]
Explicit is better than implicit.
[...]
Readability counts.
[...]
There should be one-- and preferably
only one --obvious way to do it.
```

Readability counts

Programs must be written for people to read, and only incidentally for machines to execute.

– Abelson & Sussman, “Structure and Interpretation of Computer Programs”

You can write perl in any language

```
'@'.join(['.'.join([''.join(
[chr(((ord(c)-ord('a')+13)%26)
+ord('a')) for c in w[::-1]])
for w in p.split('.')[::-1]])
for p in
'zbp.bgrvg@mpvjbuprvpwbj.gerobe'.split('@')
[::-1]])
```

Overview

- versions
  - 2.x (2.7 final)
  - 3.x (currently 3.2)
- implementations
  - CPython (C, reference/standard)
  - Jython (JVM, currently compatible with CPython 2.5)
  - IronPython (.NET, currently compatible with CPython 2.7)
  - PyPy (Python, currently compatible with CPython 2.7)
  - Stackless (C, CPython branch, microthreads, no stack)

Design

- *PEP* stands for Python Enhancement Proposal
- *PEP* is a design document providing information to the Python community, or describing a new feature for Python
- PEP-8 - coding conventions
- PEP-234 - iterators
- PEP-255 - simple generators

Coding conventions

- PEP-8 - coding conventions
- Never mix tabs and spaces
- Use 4 spaces per indentation level
- Use docstrings
- Name your classes and functions consistently
  - CamelCase for classes
  - lower_case_with_underscores for functions and methods
- Use *self* as the name for the first method argument
- Simple public attributes expose just by attribute name (properties)
- Avoid using: from module import *

Coding conventions - tabs vs spaces

```
class·Test(object):$
····def·meth1(self):$
»       »       print·'meth1'$
$
Test().meth1()$
```

```
$ python -tt test.py
[...]
TabError: inconsistent use of tabs and spaces
in indentation
$ python -m tabnanny -v test.py
'test.py': *** Line 3: trouble in tab city! ***
offending line: "\t\tprint 'meth1'\n"
indent not greater e.g. at tab sizes 1, 2
```

Module structure

```
"""module docstring"""
# imports
# constants
# exception classes
# interface functions
# classes
# internal functions & classes

def main(...):
    ...

if __name__ == '__main__':
    sys.exit(main())
```

Functions

- Variable number of arguments

```
def average(*args, **kwargs):
    lst = list(args)
    lst.extend(kwargs.values())
    return sum(lst)/float(len(lst))
```

```
>>> average(2, 3, 4, 7)
4.0
>>> average(2, 3, 4, 7, initial=9)
5.0
```

- Unpacking argument list

```
>>> args = [3, 6]
>>> range(*args)
[3, 4, 5]
```

Functions are first class objects

```
def f():
    ''' Description '''
    pass
```

```
>>> type(f)
<type 'function'>
>>> f.__name__
'f'
>>> f.__doc__
' Description '
```

Decorators

```
def trace(f):
    def wrapper(*args, **kwargs):
        print '%s: %r' % (f.__name__, args)
        ret = f(*args, **kwargs)
        print '%s: %r' % (f.__name__, ret)
        return ret
    return wrapper

def add(x, y):
    ''' Add function '''
    return x + y

>>> add_d = trace(add)
>>> add_d(2, 3)
add: (2, 3)
add: 5
5
```

Decorators (cont.)

```
@trace
def add(x, y):
    ''' Add function '''
    return x + y
```

```
>>> add(2, 3)
add: (2, 3)
add: 5
5
>>> add.__name__, add.__doc__
('wrapper', None)
```

Decorators (cont.)

```
from functools import wraps

def trace_enhanced(f):
    @wraps(f)
    def wrapper(*args, **kwargs):
        print '%s: %r' % (f.__name__, args)
        ret = f(*args, **kwargs)
        print '%s: %r' % (f.__name__, ret)
        return ret
    return wrapper
```

```
>>> add(2, 3)
add: (2, 3)
add: 5
5
>>> add.__name__, add.__doc__
('add', ' Add function ')
```

Decorators with arguments

```
def repeat(n):
    def repeat_ntimes(f):
        def wrapper(*args, **kwds):
            for i in range(n):
                ret = f(*args, **kwds)
            return ret
        return wrapper
    return repeat_ntimes

@repeat(3)
def bar():
    print 'Bar function'

>>> bar()
Bar function
Bar function
Bar function
```

DATA MODEL

Python is not Java nor C++

Reset your brain

Objects

- All data is represented by objects
- All Python objects have this
    - unique identity (an integer, returned by id(x))
    - type (returned by type(x))
    - some content

Objects (2)

- You cannot change the identity
- You cannot change the type
- Some objects allow you to change their content (mutable)
- Some objects don't allow you to change their content (immutable)
- Objects may also have this
  - Zero or more methods (provided by the type object)
  - Zero or more names

Names

- Names are not really properties of the object
- An object can have any number of names, or no name at all
- Names live in namespaces
  - module namespace
  - instance namespace
  - function local namespace

Data types

- Numbers: int, long, float, complex
- Sequences
  - immutable: string, unicode, tuple
  - mutable: list, bytearray
- Sets: set, frozenset
- Mappings: dictionary
- Functions
- Classes
  - Classic classes
  - New-style classes
- Modules

EXECUTION MODEL

Execution model

- Everything is runtime
  - Some things are cached (.pyc files)
- Execution happens in namespaces
  - Modules, functions, classes all have their own
- Modules are executed top-to-bottom
  - Just like a script
- *def* and *class* statements are runtime

Objects and bindings

- Names refer to objects
- Names are introduced by name binding operations
- Assignment statements modify namespaces, not objects
- Assignments create bindings
- The following are blocks: a module, a function body, and a class definition

Variables

- Variables are names, not containers
  - Everything is an object
  - Everything is a reference
  - Variables are neither
- Everything that holds anything, holds references
- Variables refer to objects
  - Namespaces map names to objects

Execution model - bindings

- ```
  >>> variable = 3
  >>> variable = 'hello'
  ```

- So hasn't variable just changed type?
  ```

Execution model - bindings

- ```
  >>> variable = 3
  >>> variable = 'hello'
  ```

- So hasn't variable just changed type?
- Of course not, variable isn't an object at all - it's a name
- ```
  >>> type(3), id(3)
  (<type 'int'>, 26703752)
  >>> type('hello'), id('hello')
  (<type 'str'>, 140531845285568)
  ```

Scopes

- Global

```
>>> y = 1
>>> globals()['y']
1
```

- Local

```
>>> def f():
...     x = 1
...     print locals()
...
>>> f()
{'x': 1}
```

- __builtin__

```
>>> import __builtin__
>>> dir(__builtin__)[-3:]
['vars', 'xrange', 'zip']
```

Function scope

- Definition is visible in any contained block...
- ...unless a contained block introduces a different binding for the name

```
x = 1
def g():
    print x
    x = 2
```

Class scope

- Scope of names defined in a class block is limited to the class block
- It does not extend to the code blocks of methods

```
class A(object):
    classmem = 1
    def __init__(self):
        print classmem
```

```
>>> A()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in __init__
NameError: global name 'classmem' is not define
```

Function creation

- *def* statement creates a function
- argument defaults are evaluated by function definition

Function definition

```
def func(arg1, arg2=Foo() ):
    print 'Entering func'
    def innerfunc(arg3=arg2):
        print 'Entering innerfunc'
        return arg1, arg3
    arg1 = arg2 = None
    return innerfunc
```

Function evaluation

```
def func(arg1, arg2=Foo()):
    print 'Entering func'
    def innerfunc(arg3=arg2):
        print 'Entering innerfunc'
        return arg1, arg3
    arg1 = arg2 = None
    return innerfunc
```

Class creation

- *class* statement executes block of code
  - in a separate namespace (a dict)
- class body is a normal code block

Class definition

```
class SomeClass( Parent1, Parent2 ):
    classattr = func()
    def method(self, arg):
        self.instanceattr = arg
    def helper(arg):
        return arg + 10
    for i in range(10):
        classattr = helper(classattr)
    del helper
```

Iterator protocol

- Look at the *for* statement

```
for x in obj:
    # statements
```

- Internally it uses iterator protocol

```
_iter = iter(obj) # Get iterator object
while 1:
    try:
        x = _iter.next() # Get next item
    except StopIteration: # No more items
        break
    # statements
    ...
```

- Any object that supports iter() is said to be “iterable”.

Iterator protocol - iterable

container.__iter__() - Return an iterator object

```
>>> lst = [1, 2, 3]
>>> type(lst)
<type 'list'>
>>> [hasattr(lst, f) for f in
... ('__iter__', 'next')]
[True, False]
>>> lstiter = iter(lst)
>>> type(lstiter)
<type 'listiterator'>
>>> [hasattr(lstiter, f) for f in
... ('__iter__', 'next')]
[True, True]
```

Iterator protocol - iterator

iterator.__iter__()
Return the iterator object itself.
This is required to allow
both containers and iterators
to be used with the for and in statements.

```
for line in file:
    ...
```

is a shorthand for

```
for line in iter(file.readline, ''):
    ...
```

Iterator protocol - iterator (cont.)

iterator.next()

Return the next item from the container.
If there are no further items,
raise the StopIteration exception.

```
>>> lst = [1, 2]
>>> lstiter = iter(lst)
>>> lstiter.next()
1
>>> lstiter.next()
2
>>> lstiter.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Iterator protocol - implementation

```
class MyIterable(object):
    def __init__(self):
        self._lst = [1, 2, 3]
        self._i = 0
    def __iter__(self):
        return self
    def next(self):
        if self._i < len(self._lst):
            item = self._lst[self._i]
            self._i += 1
            return item
        raise StopIteration

>>> for i in MyIterable():
...     print i,
...
1 2 3
```

Itertools module

- *itertools* module includes a set of functions for working with iterable data sets
- Inifnite iterators
  - *count* : count(10) –> 10 11 12 13 14 ...
  - *cycle* : cycle('ABCD') –> A B C D A B C D ...
  - *repeat* : repeat(10, 3) –> 10 10 10
- Iterators combining iterables
  - *chain* : chain('ABC', 'DEF') –> A B C D E F
  - *ifilter* : ifilter(lambda x: x%2, range(10)) –> 1 3 5 7 9
- Combinatoric generators
  - *permutations* : permutations('ABC', 2) –> AB AC BA BC CA CB

Generators

Generator is a simple way for creating iterators

```
>>> def gen():
...     yield 'one'
...     yield 'two'
...
>>> g = gen()
>>> type(g)
<type 'generator'>
>>> [hasattr(g, f) for f in
... ('__iter__', 'next')]
[True, True]
```

Generators (cont.)

```
def show():
    print "I am here"
    yield "Hello!"
    print "now here"
    yield "Bye!"

>>> g = show()
>>> g.next()
I am here
'Hello!'
>>> g.next()
now here
'Bye!'
>>> g.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Generators - infinite sequences

```
def iterate():
    i = 0
    while True:
        yield i
        i += 1
```

```
>>> zip(iterate(), "abc")
[(0, 'a'), (1, 'b'), (2, 'c')]
```

Generator expression

And expression that returns an iterator

```
>>> squares = (i*i for i in range(10))
>>> type(squares)
<type 'generator'>
>>> sum(squares)
285
>>> sum(i*i for i in range(10))
285
>>> squares = (i*i for i in range(10))
>>> list(squares)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Context manager protocol

with supports concept of a runtime context
defined by context mananger

```
class ContextManager(object):
    def __enter__(self):
        '''The value returned by this method
        is bound to the identifier
        in the as clause of with statements
        using this context manager.'''
    def __exit__(self, exc_type, exc_val,
                 exc_tb):
        '''Exit the runtime context
        and return a Boolean flag indicating
        if any exception that occurred
        should be suppressed.'''
```

Context manager - example

```
class FileUpperSimple(object):
    def __init__(self, fname):
        self.fname = fname
        self.__f = None
    def read(self):
        return self.__f.read().upper()
    def __enter__(self):
        self.__f = open(self.fname)
        return self
    def __exit__(self, exc_type, exc_val,
                exc_tb):
        self.__f.close()
        return False

>>> with FileUpperSimple(fname) as file_:
...     print file_.read()
```

Modules

- Module is a file with suffix “.py” containing Python definitions
- Compiled module is a file with suffix “.pyc” (or “.pyo” when -O option is used)
- Module search path
  - Directory containing the input script
  - PYTHONPATH when set (the same syntax as shell PATH)
  - sys.path initialized depending on above settings and installation default paths (e.g. /usr/lib/python2.7/)

Customizing search path: PYTHONPATH

```
$ ls hello.py*
ls: nie ma dostępu do hello.py*: Nie ma takiego
$ python -c "import hello"
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ImportError: No module named hello
$ ls test/hello.py*
test/hello.py  test/hello.pyc
$ PYTHONPATH=test python -c "import hello"
hello, World !!!
```

Customizing search path: sys.path

```
$ cat test_module.py
import hello
$ python test_module.py
Traceback (most recent call last):
  File "test_module.py", line 1, in <module>
    import hello
ImportError: No module named hello
$ vi test_module.py
$ cat test_module.py
import sys
sys.path.append('test')
import hello
$ python test_module.py
hello, World !!!
```

Where module was loaded from?

```
$ python -v -c "import httplib" 2>&1 \
> | grep httplib
# /usr/lib/python2.7/httplib.pyc matches \
> /usr/lib/python2.7/httplib.py
import httplib # precompiled from \
> /usr/lib/python2.7/httplib.pyc
```

CAUTION!

```
$ touch httplib.py
$ python -v -c "import httplib" 2>&1 \
> | grep httplib
import httplib # from httplib.py
# wrote httplib.pyc
```

Packages

```
oms
|-- __init__.py
|-- common
|   |-- __init__.py
|   `-- utils.py
|-- fm
|   |-- __init__.py
|   |-- fmadapter.py
|   `-- fmuigate.py
`-- pm
    |-- __init__.py
    |-- meahandler.py
    `-- pmfilefetcher.py
```

Packages (cont.)

- Package is a subdirectory with __init__.py file (possibly empty)
- Avoid using: from package import *
- If you really need use __all__ variable

```
__all__ = ['FMAdapter']
```

- Relative imports

```
from ..common import utils
```

- Dynamic import using __import__ function

```
oms = __import__('oms')
```

Tools

- Static analysis: pyflakes, pylint, pychecker
- Building: distutils, setuptools
- Installing: easy_install, pip, virtualenv

Pylint

```python
import sys

def some_fun():
    return 'hello'

variable = 1
print varible
print some_function()
```

```
$ pylint --disable=C --reports=n ex1.py
************* Module ex1
E:  7: Undefined variable 'varible'
E:  8: Undefined variable 'some_function'
W:  1: Unused import sys
No config file found, using default configurati
```

Distutils - Python Distribution Utilities

- Write a setup script
  (setup.py by convention)
- (optional) write a setup configuration file
- Create a source distribution
- (optional) create one or more built
  (binary) distributions

Distutils example

```
proj/
|-- doc
|-- main.py
|-- mypackage
|   |-- __init__.py
|   |-- data
|   |   `-- mydata.xml
|   `-- other_stuff.py
|-- setup.py
`-- test
```

```python
from distutils.core import setup
setup(name='MyProj',
      version='1.0',
      description='My awesome project',
      ...
      packages=['mypackage'],
```

Distribution

```
$ python setup.py sdist
running sdist
writing manifest file 'MANIFEST'
creating MyProj-1.0
...
creating 'dist\MyProj-1.0.zip' and adding 'MyPr

$ python setup.py bdist --help-formats
List of available distribution formats:
  --formats=rpm      RPM distribution
  --formats=gztar    gzip'ed tar file
  --formats=bztar    bzip2'ed tar file
  --formats=ztar     compressed tar file
  --formats=tar      tar file
  --formats=wininst  Windows executable install
  --formats=zip      ZIP file
```

Installation

```
$ unzip MyProj-1.0.zip
Archive:  MyProj-1.0.zip
  inflating: MyProj-1.0/main.py
...
$ python setup.py install
running install
running build
running build_py
creating build
creating build\lib
copying main.py -> build\lib
creating build\lib\mypackage
copying mypackage\other_stuff.py -> build\lib\m
copying mypackage\__init__.py -> build\lib\mypa
creating build\lib\mypackage\data
...
```

PyPI - Package Index

- URL: http://pypi.python.org/pypi/
- easy_install - python module bundled with setuptools that lets you automatically download, build, install and manage Python packages
- setuptools - collection of enhancements to the Python distutils
- pip - tool for installing and managing Python packages, it's a replacement for easy_install.
- virtualenv - tool to create isolated Python environments

IDIOMS

Idioms

- Swapping

```
b, a = a, b
```

- Unpacking

```
lst = ['John', 'Cleese']
firstname, surname = lst
```

- Reversing sequence

```
'python'[::-1]
```

- C-like printf

```
def printf(msg, *args):
    print msg % args
```

Idioms (2)

- Interpreter last expression result in _

```
>>> 1024 * 1024
1048576
>>> x = _
>>> x
1048576
```

- building dictionaries

```
>>> firstname = ['John', 'Michael']
>>> surname = ['Cleese', 'Palin']
>>> dict(zip(firstname, surname))
{'John': 'Cleese', 'Michael': 'Palin'}
```

- indexing collections

```
>>> items = 'zero one two'.split()
>>> list(enumerate(items))
[(0, 'zero'), (1, 'one'), (2, 'two')]
>>> for index, item in enumerate(items):
```

Idioms (3)

- Script vs module

```
if __name__ == '__main__':
```

- EAFP (Easier to Ask for Forgiveness than Permission)

```
try:                                # NO
    return mapping[key]             isinstance(x, str)
except KeyError:                    # YES
    return None                     str(x)
```

- LBYL (Look Before You Leap)

```
if key in mapping: return mapping[key]
```

Comparisons

- Comparison

```
x = 20
# NO
if x > 10 and x <= 20:
# YES
if 10 < x <= 20:
```

- Object type comparisons
  - Yes: if isinstance(obj, int):
  - No: if type(obj) is type(1):
- Empty sequences are false
  - Yes: if not seq:
  - No: if len(seq) == 0:

Batteries included

- Don’t reinvent the wheel

```
# YES
os.path.join(dname, fname)
# NO
dname + '/' + fname
```

- ```
  $ python -m SimpleHTTPServer
  Serving HTTP on 0.0.0.0 port 8000 ...
  ```
  ```

GOTCHAS

One-element tuple creation

```
>>> x = (1)
>>> x
1
>>> type(x)
<type 'int'>
>>> x = 1,
>>> x
(1,)
>>> type(x)
<type 'tuple'>
>>> x = (1,)
>>> x
(1,)
>>> type(x)
<type 'tuple'>
```

Sorting in place

```
>>> lst = [4, 3, 1]
>>> newlst = lst.sort()
>>> newlst
>>> type(newlst)
<type 'NoneType'>
>>> lst
[1, 3, 4]
```

```
>>> lst = [4, 3, 1]
>>> newlst = sorted(lst)
>>> newlst
[1, 3, 4]
>>> type(newlst)
<type 'list'>
>>> lst
[4, 3, 1]
```

Function default parameter

- Mutable object as default parameter value

```python
def f(a, lst=[]):
    lst.append(a)
    return lst

>>> f(1)
[1]
>>> f(2)
[1, 2]
>>> f(3)
[1, 2, 3]
```

```python
def f(a, lst=None):
    if lst is None:
        lst = []
    lst.append(a)
    return lst

>>> f(1)
[1]
>>> f(2)
[2]
>>> f(3)
[3]
```

Gotchas (cont.)

- Scope and variables

```
x = 1
def g():
    print x
    x = 2
```

- * operator copies references, not copies of objects

```
# NO
[[0] * 3] * 3
# YES
[[0 for _ in range(3)] for _ in range(3)]
```

```
>>> a = [[0] * 3] * 3
>>> a[0][0] = 1
>>> a
[[1, 0, 0], [1, 0, 0], [1, 0, 0]]
```

UNICODE

Unicode

- *strings* describe bytes
- *unicode* (-objects) describe characters
- Unicode can only be stored in encoding
  - ascii, latin-1, utf-8, utf-16 are encodings
- To get characters from bytes: decode
- To get bytes from characters: encode
- utf-8 is not unicode, it's unicode encoding

Unicode (2)

- Unicode literals:

```
u'\xc4 \u30c4 \u000020ac'
```

- *unichr()* instead of *chr()*
- Unicode names:

```
u'\N{EURO SIGN}'
```

- *unicodedata* module for runtime lookups
- *decode* and *encode* are generalized
  - they do more than just unicode conversion

Unicode (3)

```
>>> sys.getdefaultencoding()
'ascii'
>>> u'zażółć'.encode()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'ascii' codec can't encode

>>> sys.stdin.encoding
'UTF-8'
>>> 'zażółć'.decode('utf-8')
u'za\u017c\xf3\u0142\u0107'

>>> unicodedata.name(u'ą')
'LATIN SMALL LETTER A WITH OGONEK'
```

Unicode (4)

- Never mix unicode and bytestrings
- Decode bytestrings on input
- Encode unicode on output
- Automatic conversions help
  - codecs.open() instead of open()
  - wrap existing streams in rewriters: codecs.getreader(), codecs.getwriter()
- Pay close attention to exceptions
  - *UnicodeDecodeError* from str.encode()

Literals

- Encoding declarations
  - `# -*- coding: <encoding-name> -*-` (e.g.: `#-*- coding: utf-8 -*-`)
  - SyntaxError: Non-ASCII character ...

OBJECT ORIENTED PROGRAMMING

Classes

- Classic classes

```
>>> class Classic: pass
...
>>> classicobj = Classic()
>>> type(classicobj)
<type 'instance'>
>>> dir(classicobj)
['__doc__', '__module__']
```

- New-style classes

```
>>> class NewStyle(object): pass
...
>>> obj = NewStyle()
>>> type(obj)
<class '__main__.NewStyle'>
>>> dir(obj)
['__class__', '__delattr__', '__dict__',
...
```

Operator overloading

- There are many special methods / hooks which can be overloaded
- Numeric type:
  __add__, __sub__, __mul__ etc.
- Container type:
  __len__, __getitem__, __iter__ etc.
- Callable:
  __call__
- Attribute access:
  __getattr__, __setattr__, __delattr__

Basic but useful customization

- object.__init__(self)
  Instance initialization
- object.__str__(self)
  Called by *str* function and *print* statement to compute “informal” string representaion of an object
- object.__repr__(self)
  Called by the *repr* function to compute the “official” string representation of an object

Attribute access - adapter design pattern

```
class FTPAdapter(object):
    def __init__(self, ftpserver):
        self._ftpserver = ftpserver

    def run(self):
        self._ftpserver.start()

    def shutdown(self):
        self._ftpserver.stop()

    def __getattr__(self, attr):
        '''Everything else is delegated
           to the ftpserver'''
        return getattr(self._ftpserver, attr)
```

Factory

```
DBKIND = dict(sqlite=SQLite, oracle=Oracle)

def database(url):
    db_type, rest = url.split('://', 1)
    return DBKIND[db_type](rest)


>>> database('sqlite:///home/rw/test.db')
SQLite chosen: /home/rw/test.db
<__main__.SQLite object at 0x7fae6c1d6f90>
>>> database('oracle://user:password@tns_name')
Oracle chosen: user:password@tns_name
<__main__.Oracle object at 0x7fae6c1e6190>
```

Duck-typing

- As Fredrik Lundh has stated, "if you don’t understand duck typing, you don’t really understand Python".
- Code can be written to work with any kind of object as long as it has a certain set of methods
- Loose coupling of program components
- One of the most common examples is file-like object defined in standard library

Duck-typing - iterator

```
class Iterator(object):
    def __iter__(self):
        '''Return iterator object'''
    def next(self):
        '''Return next item'''

class Iterable(object):
    def __iter__(self):
        '''Return iterator object'''
```

Duck-typing - file-like

```python
class IFileLike(Iterator, ContextManager):
    def read(self):
        '''Read from file'''
    def readline(self):
        '''Read line from file'''
    def write(self, s):
        '''Write string to file'''
    def close(self):
        '''Close file'''
```

Property

- built-in function:
  `property([fget[, fset[, fdel[, doc]]]])`
- Return a property attribute
  for new-style classes
- Arguments
  - *fget* - function for getting an attribute value
  - *fset* - function for setting an attribute value
  - *fdel* - function for deleting an attribute
  - *doc* - docstring of the property attribute

Property - example

```
class C(object):
    def __init__(self):
        self._x = None
    def getx(self):
        return self._x
    def setx(self, value):
        self._x = value
    def delx(self):
        del self._x
    x = property(getx,setx,delx,"I'm the 'x'.")
```

```
>>> c = C()
>>> c.x = 99
>>> c.x
99
```

```
>>> C.x.__doc__
"I'm the 'x'."
```

Property - decorators

```python
class CDecor(object):
    def __init__(self):
        self._x = None

    @property
    def x(self):
        """I'm the 'x'."""
        return self._x

    @x.setter
    def x(self, value):
        self._x = value

    @x.deleter
    def x(self):
        del self._x
```

```
>>> c = CDecor()
>>> c.x = 99
>>> c.x
99
>>> help(CDecor.x)
Help on property:

    I'm the 'x'.
```

CONCURRENCY

Threading module

Python threads are defined by a class

```
class CountdownThread(threading.Thread):
    def __init__(self, count):
        threading.Thread.__init__(self)
        self.count = count
    def run(self):
        while self.count > 0:
            print 'Counting down', self.count
            self.count -= 1
            time.sleep(5)

>>> ct = CountdownThread(3)
>>> ct.start()
Counting down 3
Counting down 2
Counting down 1
```

Threading module (cont.)

Alternatively thread can be launched using function

```
def countdown(count):
    while count > 0:
        print 'Counting down', count
        count -= 1
        time.sleep(5)


>>> ct = threading.Thread(target=countdown,args
>>> ct.start()
Counting down 3
Counting down 2
Counting down 1
```

Threads execution

- Use *start* method to launch thread
- Use *join* method to wait for a thread to exit
- Threads running forever can be made daemonic

```
t.daemon = True
t.setDaemon(True)
```

Synchronization - intro

- Creating threads is really easy
- Programming with threads is hard
- Really hard

Q: Why did the multithreaded chicken cross the road?
A: to To other side. get the

– Jason Whittington

Accessing shared data

Is this actually a real concern?

```
x = 0                # A shared value
def foo():
      global x
      for i in xrange(10**6): x += 1
def bar():
      global x
      for i in xrange(10**6): x -= 1
t1 = threading.Thread(target=foo)
t2 = threading.Thread(target=bar)
t1.start(); t2.start()
t1.join(); t2.join()    # Wait for completion
print x                 # Expected result is 0
```

Yes, the print produces a random value each time:

42607, -46740, 261876

Synchronization options

- Lock - primitive mutex lock (non-reentrant)
- RLock - reentrant mutex lock
- Semaphore - counter-based synchronization primitive
- BoundedSemaphore - checks to make sure its current value doesn't exceed its initial value.
- Event - wrapper for condition variable
- Condition - condition variable

Mutex locks

- There are two basic operations:
  - ```
    m.acquire() # Acquire the lock
    m.release() # Release the lock
    ```
- Only one thread can successfully acquire the lock at any given time
- reentrant lock - it can be reacquired multiple times by the same thread
- Commonly used to enclose critical sections
    ```

Lock management

- Acquired locks must always be released
- Always try to follow this prototype

```
m = threading.Lock()
m.acquire()
try:
    do_stuff_requiring_lock()
finally:
    m.release()
```

- In Python 2.6+ it can be expressed like this

```
m = threading.Lock()
with m:
    do_stuff_requiring_lock()
```

Semaphores

- Usage

```
m = threading.Semaphore(5)  # Create
m.acquire()                 # Acquire
m.release()                 # Release
```

- *acquire()* - waits if the count is 0, otherwise decrements the count and continues
- *release()* - increments the count and signals waiting threads (if any)
- Unlike locks, *acquire()/release()* can be called in any order and by any thread

Events

- Usage

```
e = threading.Event()
e.isSet()          # Return True if event set
e.set()            # Set event
e.clear()          # Clear event
e.wait()           # Wait for event
```

- This can be used to have one or more threads wait for something to occur
- Setting an event will unblock all waiting threads simultaneously (if any)
- Common use: barriers, notification

Event example

Using an event to ensure proper initialization

```
init = threading.Event()
def worker():
    init.wait()      # Wait until initialized
    statements
    ...
def initialize():
    statements       # Setting up
    statements       # ...
    ...
    init.set()       # Done initializing
Thread(target=worker).start() # Launch workers
Thread(target=worker).start()
Thread(target=worker).start()
initialize()                  # Initialize
```

Condition variables

- Usage

```
cv = threading.Condition([lock])
cv.acquire()    # Acquire lock
cv.release()    # Release lock
cv.wait()       # Wait for condition
cv.notify()     # Signal one thread
cv.notifyAll()  # Signal all threads
```

- Lock is used to protect code that establishes some sort of "condition" (e.g., data available)
- Signal is used to notify other threads that a "condition" has changed state

Condition variable - example

- Common Use : Producer/Consumer patterns

```
items = []
items_cv = threading.Condition()
```

Producer thread

```
item = produce_item()
with items_cv:
    items.append(item)
    items_cv.notify()
```

Consumer thread

```
with items_cv:
    while not items:
        items_cv.wait()
    x = items.pop(0)
# Do something with x
...
```

- Here, the producer signals the consumer that it put data into the shared list

Threads and Queues

- Threaded programs are often easier to manage if they can be organized into producer/consumer components connected by queues
- Instead of "sharing" data, threads only coordinate by sending data to each other
- Think Unix "pipes" if you will...

Queue module

- Basic operations

```python
from Queue import Queue
q = Queue([maxsize]) # Create a queue
q.put(item)          # Put an item
q.get()              # Get an item
q.empty()            # Check if empty
q.full()             # Check if full
```

- While using *put/get* operations there is no need need to use other synchronization primitives

Queue - example

- Most commonly used to set up various forms of producer/consumer problems

```
from Queue import Queue
q = Queue()
```

Producer thread

```
for item in produce():
    q.put(item)
# Wait for consumer
q.join()
```

Consumer thread

```
while True:
    item = q.get()
    consume(item)
    q.task_done()
```

- Critical point: You don’t need locks here

Queue - signaling

- Queues also have a signaling mechanism

```
q.task_done() # Signal that work is done
q.join()  # Wait for all work to be done
```

- Used to determine when processing is done
- For each *get()* used to fetch a task,
  a subsequent call to *task_done()* tells
  the queue that the processing on the task
  is complete
- *join()* blocks until all items
  in the queue have been gotten
  and processed

Performance test

- Consider this CPU-bound function

```
def count(n):
    while n > 0:
        n -= 1
```

- Sequential execution

```
count(10**8)
count(10**8)
```

- Threaded execution

```
t1 = Thread(target=count,args=(10**8,))
t1.start()
t2 = Thread(target=count,args=(10**8,))
t2.start()
```

- Now, you might expect two threads to run twice as fast on multiple CPU cores

Bizarre results

- Performance comparison (Dual-Core Processor 2Ghz, GNU/Linux 2.6.32)
  - Sequential: 23.51s
  - Threaded: 30.69s
- Better performance without threads despite mutliple cores

Python threads implementation

- Python threads are real system threads
  - POSIX threads (pthreads)
  - Windows threads
- Fully managed by the host operating system
  - All scheduling/thread switching
- Represent threaded execution of the Python interpreter process (written in C)

The infamous GIL

- GIL - Global Intepreter Lock
- Only one Python thread can execute in the interpreter at once
- The *GIL* ensures that each thread gets exclusive access to the entire interpreter internals when it's running

GIL behaviour

- Whenever a thread runs, it holds the GIL
- However, the GIL is released
  on blocking I/O

```
  run +------+ run +------+ run +------+ run
----->+ I/O |---->| I/O |---->| I/O |---->
   \  +------+  |  +------+     +------+
    \           |      \           |
     \          |       \          |
       acquire                release
```

- So, any time a thread is forced to wait,
  other "ready" threads get their chance
  to run

How to Release the GIL

- The ctypes module already releases the *GIL* when calling out to C code
- In hand-written C extensions, you have to insert some special macros

```
PyObject *pyfunc(PyObject *self, PyObject *
    ...
    Py_BEGIN_ALLOW_THREADS
    // Threaded C code
    ...
    Py_END_ALLOW_THREADS
    ...
}
```

Why is the GIL there?

- Simplifies the implementation of the Python interpreter
- Better suited for reference counting (Python's memory management scheme)
- Simplifies the use of C/C++ extensions. Extension functions do not need to worry about thread synchronization
- And for now, it's here to stay... (although people continue to try to eliminate it)

Final Comments

- Python threads are a useful tool, but you have to know how and when to use them
  - I/O bound processing only
  - Limit CPU-bound processing to C extensions (that release the *GIL*)
- Threads are not the only way...

Processes - message passing concept

- An alternative to threads is to run multiple independent copies of the Python interpreter
- In separate processes
- Possibly on different machines
- Get the different interpreters to cooperate by having them send messages to each other

Message passing

```
+---------+ send()      recv()  +---------+
| Python  |-------------------> | Python  |
+---------+    pipe/socket      +---------+
```

- On the surface, it's simple
- Each instance of Python is independent
- Programs just send and receive messages
- Two main issues
-
    - What is a message?
    - What is the transport mechanism?

Messages

- A message is just a bunch of bytes (a buffer)
- A "serialized" representation of some data
- Creating serialized data in Python is easy

Pickle module

- A module for serializing objects
- Serializing an object onto a "file"

```
import pickle
...
pickle.dump(someobj,f)
```

- Unserializing an object from a file

```
someobj = pickle.load(f)
```

- Here, a file might be a file, a pipe, a wrapper around a socket, etc.

Pickle Commentary

- Almost any Python object can be serialized
  - Builtins (lists, dicts, tuples, etc.)
  - Instances of user-defined classes
  - Recursive data structures
- Exceptions
  - Files and network connections
  - Running generators, etc.

Message transport

- Python has various low-level mechanisms
  - Pipes
  - Sockets
  - FIFOs
- Libraries provide access to other systems
  - MPI
  - XML-RPC
  - JSON-RPC
  - Pyro (Python Remote Objects)
  - (and many others)

Pipes and Pickle

- Most programmers would use the subprocess module to run separate programs and collect their output (e.g. system commands)
- However, if you put a pickling layer around the files, it becomes much more interesting
- Becomes a communication channel where you can send just about any Python object

Multiprocessing module

- A new library module added in Python 2.6
- This is a module for writing concurrent Python programs based on communicating processes
- A module that is especially useful for concurrent CPU-bound processing

Using multiprocessing

- Here's the cool part...
- You already know how to use multiprocessing
- At a very high-level, it simply mirrors the thread programming interface
- Instead of "Thread" objects, you now work with "Process" objects.

Multiprocessing example

- Define tasks using a Process class

```
import time
import multiprocessing as mp

class CountdownProcess(mp.Process):
    def __init__(self, count):
        mp.Process.__init__(self)
        self.count = count
    def run(self):
        while self.count > 0:
            print 'Counting down', self.cou
            self.count -= 1
            time.sleep(5)
```

- You inherit from Process and redefine *run()*

Launching processes

- To launch, same idea as with threads

```
if __name__ == '__main__':
    cp = CountdownProcess(3)
    cp.start()
```

- Processes execute until run() stops
- A critical detail : Always launch in main as shown (required for Windows - no fork)

Functions as processes

- Alternative method of launching processes

```
import time
import multiprocessing as mp

def countdown(count):
    while count > 0:
        print 'Counting down', count
        count -= 1
        time.sleep(5)

if __name__ == '__main__':
    cp = mp.Process(target=countdown,args=(
    cp.start()
```

- Creates a Process object, but its *run()* method just calls the given function

Does it work?

- Consider this CPU-bound function

```
def count(n):
    while n > 0:
        n -= 1
```

- Sequential execution:

```
count(10**8)
count(10**8)
```

22.37s

- Multiprocessing execution:

```
Process(target=count,
        args=(10**8,)).start()
Process(target=count,
        args=(10**8,)).start()
```

12.36s

- Yes, it seems to work

Other Process Features

- Joining a process (waits for termination)

```
p = Process(target=somefunc)
p.start()
...
p.join()
```

- Making a daemonic process

```
p = Process(target=somefunc)
p.daemon = True
p.start()
```

- Terminating a process

```
p = Process(target=somefunc)
...
p.terminate()
```

- These mirror similar thread functions

Distributed memory

- With multiprocessing, there are no shared data structures
- Every process is completely isolated
- Since there are no shared structures, forget about all of that locking business
- Everything is focused on messaging

Pipes

- A channel for sending/receiving objects

```
(c1, c2) = multiprocessing.Pipe()
```

- Returns a pair of connection objects (one for each end-point of the pipe)
- Here are methods for communication

```
c.send(obj)               # Send an object
c.recv()                  # Receive an object
c.send_bytes(buf)         # Send a buffer
c.recv_bytes([nmax])      # Receive a buffer
c.poll([timeout])         # Check for data
```

Using pipes

- The *Pipe()* function largely mimics the behavior of Unix pipes
- However, it operates at a higher level
- It’s not a low-level byte stream
- You send discrete messages which are either Python objects (pickled) or buffers

Pipe example

- A simple data consumer

```
def consumer(p1, p2):
    # Close producer's end (not used)
    p1.close()
    while True:
        try:
            item = p2.recv()
        except EOFError:
            break
        print item # Do other useful work
```

- A simple data producer

```
def producer(sequence, output_p):
    for item in sequence:
        output_p.send(item)
```

Message queues

- Multiprocessing also provides a queue
- The programming interface is the same

```
from multiprocessing import Queue

q = Queue()
q.put(item)     # Put an item on the queue
item = q.get() # Get an item from the queue
```

- There is also a joinable Queue

```
from multiprocessing import JoinableQueue

q = JoinableQueue()
q.task_done() # Signal task completion
q.join()       # Wait for completion
```

Queue example

- A consumer process

```
def consumer(input_q):
    while True:
        # Get an item from the queue
        item = input_q.get()
        # Process item
        print item
        # Signal completion
        input_q.task_done()
```

- A producer process

```
def producer(sequence,output_q):
    for item in sequence:
        # Put the item on the queue
        output_q.put(item)
```

Other features

- Multiprocessing has many other features
  - Process pools
  - Shared objects and arrays
  - Synchronization primitives
  - Managed objects
  - Connections
- Will briefly look at one of them

Process pools

- Creating a process pool

  ```
  p = multiprocessing.Pool([numprocesses])
  ```

- Pools provide a high-level interface for executing functions in worker processes
- Let's look at an example ...

Pool example

- Define a function that does some work
- Example: Compute a SHA-512 digest of a file

```
import hashlib

def compute_sha(filename):
    digest = hashlib.sha512()
    f = open(filename,'rb')
    while True:
        chunk = f.read(8192)
        if not chunk: break
        digest.update(chunk)
    f.close()
    return digest.digest()
```

- This is just a normal function (no magic)

Pool example (cont.)

- Here is some code that uses our function
- Make a dict mapping filenames to digests

```
import os

TOPDIR = "/home/rw/src/python"

digests = {}
for root, dirs, files in os.walk(TOPDIR):
    for name in files:
        path = os.path.join(root, name)
        digests[path] = compute_sha(path)
```

- Running this takes about 10s on my machine

Pool example (cont.)

- With a pool, you can farm out work
- Here's a small sample

```
p = multiprocessing.Pool(2) # 2 processes
ret = p.apply_async(compute_sha,('ex1.py',)
...
... various other processing
...
digest = ret.get() # Get the result
```

- This executes a function in a worker process and retrieves the result at a later time
- The worker churns in the background allowing the main program to do other things

Pool example (cont.)

- Make a dictionary mapping names to digests

```
TOPDIR = "/home/rw/src/python"
p = mp.Pool(2) # Make a process pool
digests = {}
for root, dirs, files in os.walk(TOPDIR):
    for name in files:
        path = os.path.join(root,name)
        digests[path] = p.apply_async(
           compute_sha, (path,)
        )
# collect results
for path, result in digests.items():
    digests[path] = result.get()
```

- This runs in about 5.6 seconds

Multiprocessing summary

- If you have written threaded programs that strictly stick to the queuing model, they can probably be ported to multiprocessing
- The following restrictions apply
  - Only objects compatible with pickle can be queued
  - Tasks can not rely on any shared data other than a reference to the queue

Alternatives

- In certain kinds of applications, programmers have turned to alternative approaches that don't rely on threads or processes
- Primarily this centers around asynchronous I/O and I/O multiplexing
- You try to make a single Python process run as fast as possible without any thread/process overhead (e.g., context switching, stack space, and so forth)

Events and asyncore

Asyncore implements a wrapper around sockets that turns all blocking I/O operations into events

```
+-------------+    from asyncore import dispatche
|s = socket()|    class MyApp(dispatcher):
|             | /---> def handle_accept(self):
|s.accept() -+-              ...
|s.connect()-+-----> def handle_connect(self):
|s.recv()----+              ...
|s.send() -- |\----> def handle_read(self):
|...        \+--             ...
+-------------+  \--> def handle_write(self):
                             ...
                   # Create a socket
                   # and wrap it
                   s = MyApp(socket())
```

Events and Asyncore

Event loop based on select/poll

```
asyncore.loop() # Run the event loop

+------------------------------------+
|           Event loop               |
| +---------+ +---------+ +---------+ |
| | socket | | socket | | socket | |
| +---------+ +---------+ +---------+ |
+-----------------+------------------+
                  |
                  v
            +-----------+
            |dispatcher |
            +-----------+
                  |
                  v
            handle_*()
```

TESTING

Unittest module

- Python’s unit testing framework
- Added in Python 2.1
- Based on PyUnit, which was based on JUnit

Unittest API

- Test is derived from class *unittest.TestCase*
- Test method names that start with "test" are automatically invoked by the framework
- Each test method is executed independently from all other methods
- *unittest.TestCase* provides a *setUp* method for setting up the fixture, and a *tearDown* method
  for doing necessary clean-up
  - *setUp* is automatically called by *TestCase* before any other test method is invoked
  - *tearDown* is automatically called by *TestCase* after all other test methods have been invoked

How to write a test

- The code we want to test

```python
def reverse(aList):
    aList.reverse()
    return aList
```

- The test

```python
import unittest
from example1 import reverse

class ReverseTests(unittest.TestCase):
    def test_normal(self):
        # can use python's normal asserts
        assert reverse([1,2,3])==[3,2,1]
        # or more robust
        # and informative unittest options
        self.assertEqual(reverse([1,2,3]),
                         [3,2,1])
```

How to run a test

- *unittest* provides several ways
- Here's one ...

```
if __name__=="__main__":
    unittest.main()
```

Helpful methods

- fail(message)
- assertEquals(x, y)
- failUnless(expression)
- failIfExpression
- assertRaises(exception,callable,*args)

Test organization

- Aggregating individual tests into test suites
  - All tests whose names start with "test"

```
def suite():
    suite = unittest.TestSuite()
    suite.addTest(
      unittest.makeSuite(test))
    return suite
```

  - A suite can also be created from individual tests

```
suiteFew = unittest.TestSuite()
suiteFew.addTest(test("testA"))
suiteFew.addTest(test("testB"))
```

Running testsuite

- unittest.TextTestRunner
  - A basic test runner implementation which prints results on standard error
  - Default mode

    ```
    unittest.TextTestRunner().run(suite)
    ```

  - Verbose mode

    ```
    unittest.TextTestRunner(verbosity=2).ru
    ```

Unittest cons

- Test cases must be defined in classes
- Test suites must be created manually
- Tests can not be tagged for simple selection

Doctest module

- In library since Python 2.1
- No need to write separate tests
- Run the function/method under test in a Python shell,
  then copy the expected results and paste them
  in the docstring that corresponds to the tested function

Doctest usage

```python
def add(a, b):
    '''Add two numbers and return teh result
    >>> add(2, 4)
    6

    Check negative values

    >>> add(-2, 2)
    0

    Check exceptions

    >>> add(1, 'a')
    Traceback (most recent call last):
      ...
    TypeError: unsupported operand type(s) for
    '''
```

Tests in a text file

```
The ``doc`` module
======================

Using ``add``
-------------------

This is an example text file in reStructuredTex
``add`` from the ``doc`` module:

    >>> from doc import add

Now use it:

    >>> add(6, 10)
    16
```

Tests in a text file (cont.)

```
$ python -m doctest -v doc.txt
...
Trying:
    add(6, 10)
Expecting:
    16
ok
1 items passed all tests:
   2 tests in doc.txt
2 tests in 1 items.
2 passed and 0 failed.
Test passed.
```

```python
import doctest
doctest.testfile('doc.txt')
```

Nose - testing framework

- *Nose* is a unit test discovery and execution package
- Advantages
  - You can write test functions
  - Automatic tests discovery and collecting
  - Plugin support
  - Very useful standard plugins (coverage, profiler, doctests etc.)
  - Test tagging and easy selection of test sets based on tags

Nose usage

- nosetests
- ```
  # A setup.py file that uses "nose" for test

  from setuptools import setup

  setup(
      # ...
      # package metadata
      # ...
      setup_requires = ['nose'],
      test_suite = 'nose.collector',
      )
  ```
  ```

Nose - useful options

- -w: Specifying the working directory
  Run tests only from specified directory
- -s: Not capturing stdout
  By default, nose captures all output
  and only presents stdout
  from tests that fail. By specifying '-s',
  you can turn this behavior off
- -v: Info and debugging output
  nose is intentionally pretty terse.
  If you want to see what tests are being run,
  use '-v'.
- Specifying a list of tests to run

```
nosetests -w simple tests/test1.py:test_b
```

The ’attrib’ plugin

- The ’attrib’ extension module lets you flexibly select subsets of tests based on test attributes
- Consider these tests

```
def test1():
    assert 1
test1.tag = 'check'
def test2():
    assert 0
```

- `nosetests -a tag=check` will run only test1

```
$ nosetests -v -a tag=check test_attr.py
test_attr.test1 ... ok
----------------------------------------
Ran 1 test in 0.000s
```

Nose fixture

Test functions may define setup and/or teardown attributes

```
db = ''
def setup_func():
    global db
    db = 'open'
def teardown_func():
    global db
    db = 'close'
@with_setup(setup_func, teardown_func)
def test():
    assert db == 'open'
```

```
$ nosetests -vs test_fixture.py
test_fixture.test ... ok
-------------------------------
Ran 1 test in 0.001s
```

Running doctests in nose

Nose scans all non-test packages for doctests

```
def multiply(a, b):
    '''
    Multiply two numbers and return the result

    >>> multiply(5, 10)
    50
    >>> multiply(-1, 1)
    -1
    '''
    return a * b
```

```
$ nosetests -v --with-doctest doc.py
Doctest: doc.multiply ... ok
-------------------------------------
Ran 1 test in 0.016s
```

XML

XML - ElementTree

- It is API implemented by these modules
  - xml.etree.ElementTree (Python 2.5+, pure python)
  - xml.etree.cElementTree (Python 2.5+, C implementation)
  - lxml.etree (third-party library, very feature-rich, extensions to API)

ElementTree API

- From Python 2.7 ElementTree in version 1.3
- Basically only two important structures
  - Element - tree node
  - ElementTree - tree

ElementTree API (2)

- Element - tree node
  - Flexible container object to store hierarchical data structures in memory
  - Can be described as a cross between a list and a dictionary
- ElementTree - tree
  - Represents an entire element hierarchy
  - Adds some extra support for serialization to and from standard XML

Element properties

- *tag*: which is a string identifying what kind of data this element represents
- *attributes*: stored in a Python dictionary
- *text*: Text before first subelement. This is either a string or the value None, if there was no text.
- *tail*: Text after this element's end tag, but before the next sibling element's start tag. This is either a string or the value None, if there was no text.
- *child elements*: stored in a Python sequence

parsing XML - text and tail

```
from xml.etree import ElementTree as ET
root = ET.XML('<p class="p attrib">'
      + 'this goes into p.text'
      + '<em>this goes into em.text</em>'
      + 'this goes into em.tail</p>')
for elem in root.iter():
    print ('tag: %s, attrib: %s,'
          + 'text: %s, tail: %s') \
         % (elem.tag, elem.attrib,
           elem.text, elem.tail)

tag: p, attrib: {'class': 'p attrib'},
text: this goes into p.text, tail: None
tag: em, attrib: {},
text: this goes into em.text,
tail: this goes into em.tail
```

ElementTree most important functions

- parsing
  - *parse(source, parser=None)*
  - *iterparse(source, events=None, parser=None)*
- searching
  - *find(match)*
  - *findall(match)*
  - *findtext(match, default=None)*

ElementTree - searching

- Limited support for XPath expressions
- lxml provides full XPath 1.0 syntax through *xpath* method

ElementTree - incremental parsing

- *iterparse(source, events=None, parser=None)*
- Event-driven - available events: start, end, start-ns, end-ns
- Allows to track changes to the tree while it is being built
- Elements can be safely removed from tree as soon as processed (useful for large files)

OPTIMIZATION

Optimization rules

1. Get it right
2. Test it's right
3. Profile if slow
4. Optimise
5. Repeat from 2.

Optimization rules (2)

- Rule number one: only optimize when there is a proven speed bottleneck
- String concatenation
- Loops, map, list comprehension
- Avoiding dots in tight loops
- Local variables vs global variables
- Initializing dictionary elements
- Import statement overhead
- Function call overhead (inlining)
- Python is not C

Tools

- timeit - provides a simple way to time Python code
- cProfile - it's a C extension with reasonable overhead, recommended for most users
- profile - a pure Python module whose interface is imitated by cProfile
- hotshot - experimental C module that focused on minimizing the overhead of profiling
- pstats - statistics browser for reading and examining profile dumps

Timeit module

- Provides a simple interface for determining the execution time of Python code
- Uses a platform-specific time function to provide the most accurate time calculation
- Reduces the impact of startup or shutdown costs on the time calculation by executing the code repeatedly

Timeit usage

- From command line

```
$ python -m timeit -n 1000 -s "d={}" \
> "for i in range(1000):" "  d[str(i)] = i"
1000 loops, best of 3: 391 usec per loop
```

- From python code

```
import timeit
t = timeit.Timer('''
for i in range(1000):
    d[str(i)] = i''',
'd={}')
print t.timeit(1000)
print t.repeat(3, 1000)
```

Profiler usage

- From command line

```
python -m cProfile -o p1.prof p1.py o10k.ap
```

- From python code

```
import cProfile
from p1 import run
cProfile.run('run()', 'p1.prof')
```

Pstats - examining profiles

- From command line

```
$ python -m pstats p1.prof
Welcome to the profile statistics browser.

p1.prof% sort time
p1.prof% stats 5
```

- From python code

```
import pstats
p = pstats.Stats('p1.prof')
p.sort_stats('time').print_stats(5)
```