Python Idioms
to help you write
good code

EuroPython 2015
Bilbao, Spain

Marc-André Lemburg
Speaker Introduction

Marc-André Lemburg

- Python since 1993/1994
- Studied Mathematics
- eGenix.com GmbH
- Senior Software Architect
- Consultant / Trainer
- Python Core Developer
- Python Software Foundation
- EuroPython Society
- Based in Düsseldorf, Germany
Python Idioms for cooking better Code
Agenda

• Coding Conventions
• Common Patterns in Python
• Performance Idioms
• Coding Idioms
• Avoiding Gotchas
• Tools
• Questions
Agenda

- Coding Conventions
- Common Patterns in Python
- Performance Idioms
- Coding Idioms
- Avoiding Gotchas
- Tools
- Questions
Python Coding Conventions: The Basics
PEP 8 – Python Coding Conventions

• Document: http://www.python.org/dev/peps/pep-0008/

• Many useful tips on how to write readable Python code

• Guideline, not law :-)

• Can be used as basis for a corporate Python style guide

• Tool: pep8 package for checking PEP 8 compliance
Coding Conventions: Python Module Layout

Generic Python module structure:

- Header
- Tools
- Body
Module Layout: Header Section

• Module **Doc String**

• **Imports**
  – Python ones first
  – 3\textsuperscript{rd} party modules
  – application modules

• **Constants**
  – usually simple types (strings, integers)

• **Globals**
  – often private to the module
  – used for e.g. caches, static mappings, etc.
Module Layout: Tools Section

- **Exceptions**
  - used in the module
  - derived from standard exceptions
  - allow easily tracking origin of exceptions
  - often part of the module API

- **Helper Functions**
  - usually private to the module
Module Layout: Body Section

• **Functions**
  – usually part of the module API

• **Classes**
  – usually part of the module API

• **Module execution** (usually for testing or scripting)
  – if `__name__` == '__main__':
    main()
Module Layout: Overview

• Doc String
• Imports
• Constants & Globals
• Exceptions
• Helper Functions
• Functions & Classes
• Module execution (usually for testing or scripting)
  – if __name__ == '__main__':
    main()
Module Guidelines

• Use **descriptive names** for modules
  – try to name after the **most important** class or function

• Try not to use **executable code** at the module top-level
  – put all such code into functions to **make the module import side-effect free**
  – this is especially true for **package __init__.py modules**

• Always **use absolute imports**
We're now going to take a tour from high level to low level
Agenda

- Coding Conventions
- Common Patterns in Python
- Performance Idioms
- Coding Idioms
- Avoiding Gotchas
- Tools
- Questions
Common Patterns in Python

• Duck Typing
  – API interface counts, not type
  – “When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.” – James W. Riley

• Examples:
  – Python sequence API, iterator API, index API, mapping API, file API
  – ElementTree API for XML, Python DB-API
  – collections module provides ABCs
Common Patterns in Python

- **Facade / Adapter**
  - Make an object API compatible with another object
  - *Turn any bird into a duck*
  - API based, not class based

- **Examples:**
  - StringIO module (turn strings into file-like objects)
  - DB-API compatible modules (adapt database C APIs to a standard Python API)
Common Patterns in Python

• **Singletons**
  - Objects that only exist once
  - usually not enforced in Python
    (e.g. True, False are not protected in Py2, None is protected, all in Py3)

• **Examples:**
  - True, False, None, NotImplemented
  - Small integers
Common Patterns in Python

• Factories
  – Build objects in multiple ways
  – Python only has one __init__() method per class
  – use factory functions returning instances, or class methods to the same effect

• Examples:
  – mx.DateTime.DateTimeFrom()
  – datetime.fromtimestamp()
Common Patterns in Python

• **Wrappers / Proxies**
  - hide / control APIs
  - add information / verification to APIs
  - decorators often create wrappers (try to avoid this)

• **Examples:**
  - decorators, functools.partial, weakref, mxProxy
Common Patterns in Python

• **Callbacks**
  – functions/methods called when an event triggers
  – usually methods on a processing object (handler) or on a subclass (handler method)
  – hooks to extend / customize APIs
  – very common: runtime introspection for finding handlers

• **Examples:**
  – SAX parser, HTML parser, urllib2, threading
  – command line option processing
  – asynchronous processing
There's also a typical code evolution pattern in Python ...
Code Evolution: From scripts to functions

• Typical situation:
  – Start with a script using top-level commands
  – Turn common script sections into functions
  – Pass around common parameters (e.g. context related variables)
  – Group functions
Code Evolution: Abstracting functions to classes

• Next level:
  – Refactor function sections into methods of classes
  – Instantiate classes to drive the application
  – Provide a user interface (e.g. command line, web or GUI)
  – Add more input/output channels
  – More user customization
Code Evolution: Abstracting classes into components

- **Evolution:**
  - Create subclasses to provide more features
  - Split code into modules to more clarity
  - Group modules in packages
  - **Build loosely coupled components based on classes**
  - Add packaging for better deployment
  - Add automated tests and builds
  - Enter build-test-release cycle
  - Enter the code refactor cycle
Agenda

- Coding Conventions
- Common Patterns in Python
- Performance Idioms
- Coding Idioms
- Avoiding Gotchas
- Tools
- Questions
Performance Idioms

... when performance matters
Looping over sequences

• Different methods possible:
  – for-loop, map(), list comprehensions, generator

• Fastest:

Use list comprehensions

[op(x) for x in seq]

when operating on sequences
Joining strings

• Using + (concat)
  – Good when concatenating a few strings
  – Copies strings together

• Using ''.join()
  – Great for concatenating many strings
  – Copies strings, but only once

• Using StringIO.write(), array.fromstring() or %-formatting
  – Much slower than the above two
Exceptions

- Expensive in Python (but cheap in Python's C API)
- Exceptions should only be used for exceptional cases …

```python
# Not a good idea:
try:
    fails_often()
except ValueError:
    runs_often()
```
More performance hints

• Python wiki page:
  – https://wiki.python.org/moin/PythonSpeed/PerformanceTips

• My talk “When performance matters...”

• Examples:
  – localizing variables: x = global_x
  – localizing method lookups: m = list.append
  – dict.setdefault() and collections.defaultdict
Agenda

• Coding Conventions
• Common Patterns in Python
• Performance Idioms
• Coding Idioms
• Avoiding Gotchas
• Tools
• Questions
Deeply nested if statements in loops

• Typical situation:

```python
for x in items:
    if cond1(x):
        do_something()
    if cond2(x):
        do_something_else()
```
Deeply nested if statements in loops

- **Staircase Syndrom:**

```python
for x in items:
    if cond1(x):
        do_something()
    if cond2(x):
        do_something_else()
```
Better: Use **continue** to avoid nesting

for x in items:

    if not cond1(x):
        **continue**

    do_something()

    if not cond2(x):
        **continue**

    do_something_else()
... mutable or not, that is the question
(well, at least one of them)
Avoid **mutable state** in module globals

- **Typical situations** at module scope:
  
  ```python
  db = database.connect(...)  
  settings = dict( verbose=1, log=False)
  ```

- **Not thread safe**

  ... and no, *thread locals* are not a good idea

- **Difficult to manage**
  (access, resources, validations)

- **Dangerous in larger projects**
Better: Place mutable state in context objects

- Keep state in context objects
- Pass context to methods and functions as first parameter
  ```python
def read_data(context, key):
  def get_config(context):
  
or refactor code to use instance attributes:
  ```
class Application:
  def __init__(self):
    self.context = ApplicationContext()
Function/method default parameters

- Use only immutable types
- For mutable types:
  - fallback to None or
  - immutable alternatives
  - Add mutable defaults at the top of the function/method:

```python
def func(x, a=None):
    if a is None:
        a = []
    return x(a)
```
Class default attributes

• Only use immutable types
  – unless you really know what you're doing

• For mutable types:
  – fallback to None or
  – immutable alternatives
  – Add mutable type defaults in __init__()

• Document all attributes using comments:

  # List of file names
  files = None
Avoiding Gotchas

... and finally, some very basic hints to write better code
Use the *in* operator in Python

- Interface to the `__contains__` method of an object.
- If `x in some_list`: ...
  - Better than using `some_list.index(x)` with try-except.
- If `x in some_dict`: ...
  - Better than using `some_dict.has_key(x)`
- If `x in some_set`: ...
  - Better than ??? (only way to test membership :-))
- Function equivalent:
  
  ```python
  a in b = operator.contains(b, a)
  ```
Boolean testing in Python

• Boolean singletons:
  True (=integer 1), False (= integer 0)

• Use: if x: print(“x is true”)
  Not: if x == True: print(“x is true”)
  Not: if x is True: print(“x is true”)

• Use: if not x: print(“x is false”)
  Not: if x == False: print(“x is false”)
  Not: if x is False: print(“x is false”)

• Function equivalent: bool(x)
Container empty testing in Python

- **Empty containers** (sequences, mappings, sets) are considered false in Python

- Use: `if container: print("not empty")`

  *Not*: `if len(container): print("not empty")`

  *Not*: `if len(container) > 0: print("not empty")`
Commas in Python: 1-Tuples

- **Nasty in 1-tuples:**
  
  ```python
  x = (1,)
  x = 1,
  ← note the comma
  ```

- **Weird errors when not intended…**
  
  ```python
  >>> s = 'abc',
  >>> len(s)
  1
  >>> s[2]
  IndexError: tuple index out of range
  ```
Commas in Python: Listings

• Great in argument/parameter/item listings:

Python allows trailing “,” at end of listings...

```python
d = {'a': 1,
     'b': 2,
     # insert something new here
 }
```

even in functions/methods...

```python
f(x, y, z,
a=1,
    # insert more keyword args here
 )
```
Agenda

• Coding Conventions
• Common Patterns in Python
• Performance Idioms
• Coding Idioms
• Avoiding Gotchas
• Tools
• Questions
Tools

- The Python community has created some really good tools to help with developing good code

Let's have a look at some …
Code Checking Tools

- **pylint**: https://pypi.python.org/pypi/pylint/
  - style/typo checking
  - good documentation: http://docs.pylint.org/

- **pep8**: https://pypi.python.org/pypi/pep8/
  - style checking

- **pyflakes**: https://pypi.python.org/pypi/pyflakes/
  - logical checking
  - no documentation
Code Checking Tools

- **flake8:**
  - combines pep8, pyflakes and a complexity checker

- **pychecker:** [http://pychecker.sourceforge.net/](http://pychecker.sourceforge.net/)
  - note: imports modules
  - not maintained anymore (predates the other tools), but can still be useful
Agenda

• Coding Conventions
• Common Patterns in Python
• Performance Idioms
• Coding Idioms
• Avoiding Gotchas
• Tools
• Questions
Questions

```python
>>> raise Question()
```
Thank you for listening

Beautiful is better than ugly.
Photo References

CC BY / CC BY-SA licensed photos:
https://www.flickr.com/photos/runlevel0/6297898421/

Public domain clip arts:
https://openclipart.org/detail/8879/rubber-duck
https://openclipart.org/detail/215201/evolution

All other photos and images are used by permission.
Contact

eGenix.com Software, Skills and Services GmbH
Marc-André Lemburg
Pastor-Löh-Str. 48
D-40764 Langenfeld
Germany

eMail: mal@egenix.com
Phone: +49 211 9304112
Fax: +49 211 3005250
Web: http://www.egenix.com/