Python programming — Pythonish python

Finn Årup Nielsen

DTU Compute
Technical University of Denmark

November 4, 2013

Overview

"Pythonic": Writing Python the Python way.

Langtangen's list of Pythonic programming

Special Python constructs: class variables, generators, introspection . . .

”Pythonic”

“Code that is not Pythonic tends to look odd or cumbersome to an experienced Python programmer. It may also be overly verbose and harder to understand, as instead of using a common, recognizable, brief idiom, another, longer, sequence of code is used to accomplish the desired effect.”
— Martijn Faassen

Langtangen's list of Pythonic programming

From (Langtangen, 2008, Section B.3.2 Pythonic Programming, page 30+) in appendix

1. Make functions and modules
2. Use doc strings (and perhaps doctest and docopt)
3. Classify variables as public or non-public (the underscore)
4. Avoid indices in list access (in the for-loops)
5. Use list comprehension

6. Input data are arguments, output data are returned
7. Use exceptions (instead of if)
8. Use dictionaries
9. Use nested heterogeneous lists/dictionaries
10. Use numerical Python (Numpy will typically be faster than your for-loops)
11. Write str and repr functions in user-defined classes (for debugging)
12. Persistent data

13. Operating system interface (use cross-platform built-in functions)

Langtangen #2: The docstring . . .

You use the docstring for structured documentation for yourself and others, explaining the class, method, function, module.

Docstrings and simple testing: doctest

Docstring and automatic generation of documentation:

- Sphinx: Python documentation generation
- pydoc: Documentation generator and online help system in Python Standard Library.
- epydoc
- . . .

. . . Langtangen #2: The docstring

You can also use docstrings for specifying tests (doctest) and input arguments (doctopt), — and other strange things:

```
def afunction(**kwargs):
    """
    y = x ** 2 + a
    """
    return eval(afunction.__doc__.split("\n")[1].split("=")[1],
                globals(), kwargs)
```

Gives:

```
>>> afunction(x=2, a=1)
5
```

See a more elaborate example in Vladimir Keleshev's video How to write an interpreter by Vladimir Keleshev

Langtangen #3: Where is private?

There are no private variables in Python.

By convention use a prefix underscore to signal a private variable.

```
def _this_signals_a_private_function(*args):
    pass

der this_signals_a_public_function(*args):
    pass
```

Attempt on private variable

Example from Documention of Built-in functions:

```
class C(object):                 # Inherit from object
    def __init__(self):
        self._x = None           # 'Hidden'/private variable
    @property                    # 'property' decorator
    def x(self):
        """I'm the 'x' property."""
        return self._x
    @x.setter                    # decorator
    def x(self, value):
        self._x = value
    @x.deleter                   # decorator
    def x(self):
        del self._x
```

. . . Attempt on private variable

Use of the property:

```
>>> a = C()
>>> a.x = 4
>>> a.x
4
>>> a._x
4
>>> a._x = 5          # The "user" of the class is still allowed to set
>>> a._x              # But now it is his own fault if something breaks
5
>>> a.x
5
```

. . . Attempt on private variable

Making a property that is non-negative:

```
class C(object):
    def __init__(self):
        self._x = None            # 'Hidden'/private variable
    @property
    def x(self):
        return self._x
    @x.setter
    def x(self, value):
        self._x = max(0, value)  # Only allow non-negative values
    @x.deleter
    def x(self):
        del self._x
```

. . . Attempt on private variable

```
>>> a = C()
>>> a.x = -4              # Ordinary setting of property
>>> a.x
0
>>> a._x = -4             # Programmer abuses the object interface
>>> a.x
-4                        # The property is set to an illegal value
```

Langtangen #4: Pythonic for-loop

Non-pythonic:

```
alist = ['DTU', 'KU', 'ITU', 'CBS']
for n in range(len(alist)):
    print(alist[n])
```

More "Pythonic":

```
for university in alist:
    print(university)
```

Langtangen #5: List comprehensions . . .

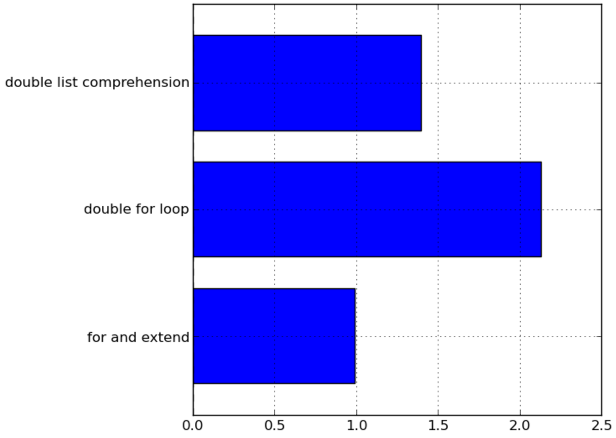
```
>>> setup = """list_of_lists = [[1], [3, 4], [6, 7, 8], [9],
                                 [10, 11, 12, 13]]"""
```

We would like to flatten and copy this.

```
code = {}
code["double for loop"] = """flatten_list = []
for a_list in list_of_lists:
    for elem in a_list:
        flatten_list.append(a_list)"""

code["for and extend"] = """flatten_list = []
for a_list in list_of_lists:
    flatten_list.extend(a_list)"""

code["double list comprehension"] = """
flatten_list = [ elem for a_list in list_of_lists
                      for elem in a_list ]"""
```

. . . Langtangen #5: List comprehensions . . .

```
from timeit import timeit
from pandas import DataFrame

def time_blocks(code):
    timings = []
    for name, block in code.items():
        timings.append((name, timeit(block, setup)))
    return DataFrame.from_dict(dict(timings), orient="index")
```



```
timings = time_blocks(code)
timings.plot(kind="barh", legend=False)
gcf().subplots_adjust(left=0.3)
show()
```

In this example list comprehension was not the fastest!

. . . Langtangen #5: List comprehensions . . .

Conditional flattening of a list of lists:

```
c1 = """flatten_list = []
for a_list in list_of_lists:
    for elem in a_list:
        if elem % 2 == 0:
            flatten_list.append(a_list)"""

c2 = """flatten_list = []
for a_list in list_of_lists:
    flatten_list.extend(filter(lambda v: v % 2 == 0, a_list))"""

c3 = """flatten_list = [ elem for a_list in list_of_lists
                              for elem in a_list if elem % 2 == 0]"""
```

. . . Langtangen #5: List comprehensions

```
>>> timeit(c1, setup)
2.4064879417419434
>>> timeit(c2, setup)
4.6115639209747314
>>> timeit(c3, setup)
1.8284208774566665
```

Here the list comprehension is the fastest.

Langtangen #11: Write str and repr functions

`__str__` is for a readable representation, `__repr__` for the “official” string representation (should look like a Python expression).

```
from numpy import matrix

class FactorizableMatrix(matrix):
    def __repr__(self):
        rows = [ ", ".join(map(str, row)) for row in A.tolist() ]
        return "FactorizableMatrix([[" + "],\n [".join(rows) + "]])"
```

Example use:

```
>>> A = FactorizableMatrix(numpy.random.rand(2,3))
>>> A
FactorizableMatrix([[0.064677281455, 0.555777048471, 0.24262937122],
 [0.435645994003, 0.0907782974028, 0.0821021379862]])
```

Langtangen #11: Write str and repr functions

Calling the `__str__` method:

```
>>> print(A)
[[ 0.06467728  0.55577705  0.24262937]
 [ 0.43564599  0.0907783   0.08210214]]
```

Here the parent (`numpy.matrix`) `__str__` method is called

Direct call:

```
>>> A.__str__()
[[ 0.06467728  0.55577705  0.24262937]\n [ 0.43564599  0.0907783   0.08210214]]'
```

Langtangen #13: Operating system interface

Use cross-platform built-in functions

Listing parent directories:

```
import os
os.system('ls -al .. ')       # 'ls' is only available on some systems

os.listdir('..')              # '..' could also be problematic

os.listdir(os.path.pardir)    # Better way
```

Also note forward and backward slash problem (cross-platform: use `os.path.sep` instead of "/") and globbing (`glob.glob('*.pdf')`)

Other Python idiosyncrasies beyond Langtangen

Instance variables vs. class (static) variables

```
class MyClass:
    my_static_variable = "Static"    # not self.
    def __init__(self):
        self.my_instance_variable = "Instance"
    def change_static(self):
        MyClass.my_static_variable = "Changed"   # not self.
    def change_instance(self):
        self.my_instance_variable = "Also changed"

my_first_instance = MyClass()
my_second_instance = MyClass()
my_first_instance.change_static()      # Will also change the second
my_first_instance.change_instance()    # instance variable
```

Instance variables vs. class (static) variables

Result:

```
>>> print(my_second_instance.my_static_variable)
Changed
>>> print(my_second_instance.my_instance_variable)
Instance
```

So the class variable is shared across instances.

Note there is also a global statement, but you can probably avoid globals using classes and class variables.

Generators

Generators can be used as pipes (computing on an infinite dataset):

```
def peak_to_peak(iterable):
    it = iter(iterable)
    first_value = it.next()
    the_min = first_value
    the_max = first_value
    while True:
        value = it.next()
        if value < the_min:
            the_min = value
            yield the_max - the_min      # Only yield when peak changed
        elif value > the_max:
            the_max = value
            yield the_max - the_min      # Only yield when peak changed


import random
def randoms():
    while True: yield random.random()    # Just get some random numbers

for peak in peak_to_peak([1, 2, 3, 5, -5, 3, 3, 8, 10, 100, 1, 1]): print(peak)
for peak in peak_to_peak(randoms()): print(peak)
```

. . . Generators

Yet another generator in the pipe:

```
def stop_at(iterable, limit=10):
    it = iter(iterable)
    while True:
       value = it.next()
       yield value
       if value > limit:
           break


def randoms():
    while True: yield random.normalvariate(0, 1)

for peak in stop_at(peak_to_peak(randoms()), 10.5):
    print(peak)
```

Introspection

You can do strange this with introspection, e.g., having data/code in the docstring that you use:

```
class AFunction():
    def __init__(self, **defaults):
        self._defaults = defaults
    def __call__(self, **kwargs):
        kwargs.update(**self._defaults)
        return eval(self.__doc__.split("\n")[1].split("=")[1],
                globals(), kwargs)

class Parabola(AFunction):
    """
    y = a * x ** 2 + b
    """
    # Look! No methods!
```

. . . Introspection

Using the derived class:

```
>>> parabola = Parabola(a=1)
>>> parabola(x=2, b=1)
5
```

It also works with Numpy:

```
>>> import numpy as np
>>> parabola(x=np.array([1, 2, 3, 4]), b=1)
array([ 2,  5, 10, 17])
```

More information

Hidden features of Python

Chapter 4. The Power Of Introspection from Dive Into Python.

Summary

Write Pythonic code.

Consider the Langtangen's list of Pythonic programming

There are a number of (non-introductory-Python-programming) details in the Python programming language that might come in handy: generators, class variables, decorators, etc.

References

Langtangen, H. P. (2008). *Python Scripting for Computational Science*, volume 3 of *Texts in Computational Science and Engineering*. Springer, Berlin, third edition edition. ISBN 978-3-642-09315-9.