LISA CCW Tutorial on Effective Python

David Warde-Farley

February 4, 2014

0.1 Python 2 vs. Python 3

Python 3 is an incompatible new major version of the Python language. It contains many new features and "fixes" to the language (e.g. integer division) that render it incompatible with Python 2, the long-standing standard.

In the lab, we use exclusively Python 2.7. We try to maintain compatibility with older versions for certain projects: Python 2.6 (Pylearn2) and Python 2.4 (Theano, due to Ubisoft). There are compatibility modules in Theano that facilitate this.

However, we do try to make code forward-compatible when possible. Theano currently works on Python 3, for example.

0.2 Python Style

Lab projects try to follow PEP8, a set of recommendations for formatting Python code.

Pylearn2 uses the NumPy docstring standard commonly employed in the scientific Python ecosystem (by projects such as NumPy itself, SciPy, Pandas, and scikit-learn). It provides readable, detailed docstrings that can nonetheless be formatted into nice-looking HTML.

Theano documentation style is somewhat less standardized.

0.3 Python Idioms

Swapping two values is easier than most languages:

```
In [1]: a = 5
        b = 6
        # Swap
        temp = a
        a = b
        b = temp
        print a, b
```

```
6 5
```

```
In [2]: a, b = b, a
        print a, b
```

```
5 6
```

Sequences can be *unpacked* into several variables without several lines of indexing:

```
In [3]: x = [1, 2, 3]
        t, u, v = x
        print t
        print u
        print v
```

```
1
2
3
```

Also in for loops:

```
In [4]: x = [(1, 2), (2, 3), (4, 3)]
        for val1, val2 in x:
            print val1, "is value 1"
            print val2, "is value 2"
```

```
1 is value 1
2 is value 2
2 is value 1
3 is value 2
4 is value 1
3 is value 2
```

You can even unpack nested sequences this way:

```
In [5]: a = ('David', (25, 50))
        c, (d, e) = a
        print c, "is c"
        print d, "is d"
        print e, "is e"
```

```
David is c
25 is d
50 is e
```

The trouble is that you must know the precise length of the thing you are unpacking (fixed in Python 3).

```
In [6]: a, b = [1, 2, 3]
```

```
        ---------------------------------------------------------------------------
    ValueError                                Traceback (most recent call last)

        <ipython-input-6-52b2a395c10f> in <module>()
    ----> 1 a, b = [1, 2, 3]


        ValueError: too many values to unpack
```

Tuples are constructed by the **comma**: parentheses are incidental.

```
In [7]: 5, 3
```

```
Out[7]: (5, 3)
```

_ is convenient for a few things.

```
In [8]: # Contains return value of last evaluated expression
        # in the interactive prompt
        _
```

```
Out[8]: (5, 3)
```

```
In [9]: # Convenient for unused values in unpacking
        q = ["David", "Warde-Farley", 16]
        _, _, num = q
        print num
```

```
16
```

0.4 Working with strings

Joining strings:

```
In [10]: # Terrible memory access patterns, can be
         # slow if you have big strings.
         x = ['a', 'b', 'c']
         y = ''
         for a in x:
             y += a
         print y
```

```
abc
```

```
In [11]: # More readable
         print ''.join(x)
```

```
abc
```

```
In [12]: # Can be any separator
         print ', '.join(x)
```

```
a, b, c
```

Formatting strings

```
In [13]: "%s is %d years old." % ("David", 29)
```

```
Out[13]: 'David is 29 years old.'
```

```
In [14]: "%(name)s is %(age)d years old." % {'name': 'David', 'age': 29}
```

```
Out[14]: 'David is 29 years old.'
```

```
In [15]: "asdfsdf {foo}".format(foo='bar')
```

```
Out[15]: 'asdfsdf bar'
```

Safety with formatting strings: use a tuple if you're not sure

```
In [16]: v = 'the string i want to add in the other string'
         v = (5, 3)
         "str: %s" % ( v,)
```

```
Out[16]: 'str: (5, 3)'
```

0.5 Working with dictionaries

Defaults when getting values

```
In [17]: # Bad
         ages = {'David': 29, 'Ian': 28}
         key = 'Guillaume'
         if key not in ages:
             val = '???'
         else:
             val = ages[key]
         print val
```

```
???
```

```
In [18]: # Simpler, much shorter, same thing
         print ages.get(key, '???')
```

```
???
```

Defaults when setting values

```
In [19]: counts = {'bread': 5}
         key = 'cheese'
         if key not in counts:
             counts[key] = 1
         else:
             counts[key] = counts[key] + 1
```

```
In [20]: counts.setdefault(key, 0)
         counts[key] += 1  # could also use defaultdict
```

```
In [21]: # d.setdefault returns value of d[key], initializing first if needed
         print "I have", counts.setdefault("bacon", 0), "bacon."
         print "I have", counts.setdefault("cheese", 0), "cheese."
```

```
I have 0 bacon.
I have 2 cheese.
```

Iteration on dictionaries:

```
In [22]: counts
```

```
Out[22]: {'bacon': 0, 'bread': 5, 'cheese': 2}
```

```
In [23]: for k in counts:
             print k
```

```
cheese
bacon
bread
```

```
In [24]: counts.keys()
```

```
Out[24]: ['cheese', 'bacon', 'bread']
```

```
In [25]: counts.values()
```

```
Out[25]: [2, 0, 5]
```

```
In [26]: counts.items()
```

```
Out[26]: [('cheese', 2), ('bacon', 0), ('bread', 5)]
```

Use iterator methods where possible:

```
In [27]: for k, v in counts.iteritems():
             print k, 'is the key'
             print v, 'is the value'
```

```
cheese is the key
2 is the value
bacon is the key
0 is the value
bread is the key
5 is the value
```

```
In [28]: counts.iteritems() # Not a list! Lazily evaluated == less memory.
```

```
Out[28]: <dictionary-itemiterator at 0x3108d08>
```

Beware! Dictionaries are a source of non-determinism.

```
In [29]: from collections import OrderedDict
         counts2 = OrderedDict()
         counts2['cheese'] = 2
         counts2['bread'] = 1
         print counts2
```

```
OrderedDict([('cheese', 2), ('bread', 1)])
```

0.6 Iteration more generally

`zip()` is your friend:

```
In [30]: names = ['Ian', 'David', 'Mehdi']
         foods = ['cheese', 'bread', 'salsa']
         zip(names, foods, foods, names)
```

```
Out[30]: [('Ian', 'cheese', 'cheese', 'Ian'),
          ('David', 'bread', 'bread', 'David'),
          ('Mehdi', 'salsa', 'salsa', 'Mehdi')]
```

If you're iterating over very large collections of things, `izip` is preferable

```
In [31]: from itertools import izip
         for n, f in izip(names, foods):
             print n, f
```

```
Ian cheese
David bread
Mehdi salsa
```

```
In [32]: izip(names, foods) # Not a list! Lazily evaluated.
```

```
Out[32]: <itertools.izip at 0x310a2d8>
```

Zipping with a range? There's a builtin called `enumerate` for that.

```
In [33]: for i, name in enumerate(names):
             print i, name

0 Ian
1 David
2 Mehdi
```

If you make friends with `itertools`, it'll never let you down.

```
In [34]: from itertools import product

         for a, b, c in product(names, foods, foods):
             print a, b, c

Ian cheese cheese
Ian cheese bread
Ian cheese salsa
Ian bread cheese
Ian bread bread
Ian bread salsa
Ian salsa cheese
Ian salsa bread
Ian salsa salsa
David cheese cheese
David cheese bread
David cheese salsa
David bread cheese
David bread bread
David bread salsa
David salsa cheese
David salsa bread
David salsa salsa
Mehdi cheese cheese
Mehdi cheese bread
Mehdi cheese salsa
Mehdi bread cheese
Mehdi bread bread
Mehdi bread salsa
Mehdi salsa cheese
Mehdi salsa bread
Mehdi salsa salsa
```

A `product` call is a good deal easier to read than several nested loops.

0.7 Transforming sequences

List comprehensions:

```
In [35]: capital_names = []
         for name in names:
             capital_names.append(name.upper())
         print capital_names

['IAN', 'DAVID', 'MEHDI']
```

```
In [36]: # More readable, one line
         capital_names = [name.upper() for name in names]
         print capital_names
```

```
['IAN', 'DAVID', 'MEHDI']
```

Generator expressions:

```
In [37]: ':'.join(name.upper() for name in names)  # Same as list comp, no []
```

```
Out[37]: 'IAN:DAVID:MEHDI'
```

Generators more generally:

```
In [38]: def mygen():
             yield 1
             print "abc"
             yield 2
             print "def"
             yield 3

         for i in mygen():
             print "i =", i
```

```
i = 1
abc
i = 2
def
i = 3
```

Beware! Generator objects are not serializable by `pickle`, the Python native serialization format for objects. But you can build iterable objects that can be serialized. Look at `pylearn2/utils/iteration.py` for examples.

0.8 Sorting

In general, common operations on built-in types are available by methods or built-in top-level functions. Sometimes both!

```
In [39]: l = [5, 6, 3]
         l.sort()
         print l
         l.extend([8, 9, 10])
         print l
```

```
[3, 5, 6]
[3, 5, 6, 8, 9, 10]
```

You can also sort with different keys, by providing a callable that produces the right key. The `operator` module contains some basic ones.

```
In [40]: name_age_pairs = [('David', 29), ('Ian', 28)]
         import operator

         name_age_pairs.sort(key=operator.itemgetter(1))
         print name_age_pairs
```

```
[('Ian', 28), ('David', 29)]
```

NumPy also has sorting capabilities.

```
In [41]: import numpy as np
         myarr = np.array([9, 6, 7, 8])
         argsorted = np.argsort(myarr)
         myarr[argsorted]

Out[41]: array([6, 7, 8, 9])
```

0.9 Command line processing

`argparse` is now in the standard library. Provides GNU-style command-line argument processing and automatic help screen generation. Don't underestimate the value of this, you'll thank your past self for writing a good `--help` when you have to understand how to use a script you wrote 6 months ago.

0.10 The filesystem

```
In [42]: path_components = ['path', 'to', 'some', 'file.txt']
         print '/'.join(path_components) # Fail on Windows!

         import os.path
         print os.path.join(*path_components) # Cross plaform!
         # Also os.path.split for same issue.
```

```
path/to/some/file.txt
path/to/some/file.txt
```

By the way, the * syntax above "unpacks" the tuple into the argument list.

```
In [43]: def f(a, b):
             print a + b

         x = [5, 3]

         f(x[0], x[1])
         f(*x)
         d = {'a': 5, 'b': 3}
         f(**d)
```

```
8
8
8
```

The preferred way of doing file I/O since Python 2.6:

```
In [44]: with open('/dev/null') as f:
             # This syntax always cleans up the file object f,
             # i.e. f.close(), even if exception
             pass

         # Old, bad way:
         f = open('/dev/null')
         # What happens if an exception happens here?
         f.close()
```

The with statement works with "Context managers", look them up if you want to know how to make your own.

0.11 Odds and ends

Testing for multiple values

```
In [45]: key = 'David'
         # NO
         if key == 'Guillaume' or key == 'Ian' or key == 'Fred':
             pass

         # YES
         if key in ('Guillaume', 'Ian', 'Fred'):
             pass

         # if this was really big... use a set
         allowed_users = set(('Guillaume', 'Ian', 'Fred'))
         if key in allowed_users:
             pass
```

Sets can be useful in this and other contexts:

```
In [46]: a = set([5, 3, 6])
```

Note that unlike other containers (lists, tuples), you can't index into a set, as sets are inherently unordered.

```
In [47]: a[0]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)

<ipython-input-47-5ccf417d7af1> in <module>()
----> 1 a[0]


TypeError: 'set' object does not support indexing
```

But you can test containment, just like lists and tuples, but generally faster.

```
In [48]: 5 in a

Out[48]: True

In [49]: 7 in a

Out[49]: False
```