CS506/CS606 (Research Programming)
Python style & tooling

Kyle Gorman
Center For Spoken Language Understanding
Oregon Health & Science University

1 Python style

- PEP 8 (`http://legacy.python.org/dev/peps/pep-0008/`) is a detailed style guide for Python endorsed by the core developers. Here are some key clauses (here presented verbatim):
  - Use 4 spaces per indentation level.
  - Limit all lines to a maximum of 79 characters.
  - Separate top-level function and class definitions with two blank lines. Method definitions inside a class are separated by a single blank line.
  - Avoid extraneous whitespace in the following situations:
    * Immediately inside parentheses, brackets, or braces.

      ```
      Yes: spam(ham[1], {eggs: 2})
      No:  spam( ham[ 1 ], { eggs: 2 } )
      ```

    * Immediately before a comma, semicolon, or colon.

      ```
      Yes: if x == 4: print x, y; x, y = y, x
      No:  if x == 4 : print x , y ; x , y = y , x
      ```

    * Immediately before the open parenthesis that starts the argument list of a function call.

      ```
      Yes: spam(1)
      No:  spam (1)
      ```

    * Immediately before the open parenthesis that starts an indexing or slicing.

      ```
      Yes: dct['key'] = lst[index]
      No:  dct ['key'] = lst [index]
      ```

    * More than one space around an assignment (or other) operator to align it with another.

- To automatically check your adherence to PEP 8, use the `pep8` command.

```
$ pep8 detectormorse.py
detectormorse.py:36:15: E128 continuation line under-indented
    for visual indent
detectormorse.py:43:43: W291 trailing whitespace
...
```

- To attempt to automatic fixes to these violations, use the `autopep8` command; the `-i` flag causes it to change the script(s) in place.

```
$ autopep8 -i detectormorse.py
```

- When in need of further inspiration, consult the Zen of Python.

```
$ python -m 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.
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!
```

2 Python tooling

- Python ships with a suite of tools for debugging, unit testing, profiling, and timing. These are usually made available to the user in three ways:

1. As modules that can be imported and controlled within a user's script; this provides the greatest level of control, though it is more verbose
2. As modules executed from the command line using `python -m ...`
3. In IPython via `%magic`, as demonstrated below

2.1 Debugging with `pdb`

- The `pdb` module is a traditional built-in interactive debugger. It can be invoked from the command line by calling `python -m pdb` followed by a normal invocation of your script.

```
$ python -m pdb detectormorse.py detectormorse.py -t trn.txt
   -e tst.txt
> /Users/gormanky/.../detectormorse.py(26)<module>()
-> import logging
(Pdb) next
> /Users/gormanky/.../detectormorse.py(26)<module>()
-> from nlup.decorators import IO
...
```

- To manually add breakpoints to your script, simply import the `pdb` module and execute `pdb.set_trace()` anywhere you want to break into the debugger.

- In IPython, the `%pdb` command enables automatic debugging, which drops the user into interactive debugging anytime an exception is thrown.

```
In [1]: %pdb
Automatic pdb calling has been turned ON

In [2]: raise ValueError
---------------------------------------------
ValueError
Traceback (most recent call last)
<ipython-input-2-94ef6d30a139> in <module>()
----> 1 raise ValueError

ValueError:
> <ipython-input-2-94ef6d30a139>(1)<module>()
----> 1 raise ValueError
ipdb>
...
```

2.2 Testing with `doctest`

- *Unit tests* are a powerful tool for preventing undetected *code regressions*. In Python, the `doctest` module allows unit tests to be mixed in with documentation strings at the function,

class, or module level. These tests may act as examples of a typical invocation of the code being tested.

- To mark a line of a documentation string as a doctest, simply prefix it with >>>, and put the expected return value on the next line.

```
def square(x):
    """
    Return `x` squared

    >>> square(3)
    9
    >>> square(9)  # this is intentionally wrong
    100
    """
    return x * x
```

When the tests are run, the output of the >>> statement is (string)-compared against the next line; if they do not match, an error message is generated.

```
**********************************************
File "square.py", line 5, in square.square
Failed example:
    square(9)  # this is intentionally wrong
Expected:
    100
Got:
    81
**********************************************
1 items had failures:
   1 of   2 in square.square
***Test Failed*** 1 failures.
```

- There are two ways to run doctests:

  1. Calling `doctest.testmod()` in `__main__` will run all doctests in the current module.
  2. At the command line, `python -m doctest ...` will run doctests in any modules given as arguments.

2.3 Profiling with `cProfile`

- *Profiling* provides a global view of execution time. The `cProfile` module computes for each function the number of calls and execution times during a single run of that script. This allows the developer to determine which components of the runtime might benefit from optimization. To run the profiler, simply invoke the module at the command line, followed by the name of the script and any arguments to it. Use the `-s` flag before the script name to specify a sorting

rule for the functions profiled; a useful one is cumulative time spent within each function (`-s cumulative`).

```
$ python -m cProfile -s cumulative detectormorse.py -t trn.
   txt -e tst.txt
...
         55605510 function calls (55595653 primitive calls)
            in 50.721 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno
      (function)
    651/1    0.039    0.000   50.723   50.723 {built-in
       method exec}
        1    0.001    0.001   50.723   50.723 detectormorse.
           py:24(<module>)
        1    0.012    0.012   42.004   42.004 detector.py:84(
           __init__)
        1    0.137    0.137   41.991   41.991 detector.py
           :190(fit)
    61925    1.071    0.000   26.273    0.000 detector.py:97(
       candidates)
   123848    0.133    0.000   22.579    0.000 __init__.py:86(
      word_tokenize)
        1    0.970    0.970   18.410   18.410 perceptron.py
           :62(fit)
   123848    0.146    0.000   16.600    0.000 __init__.py
      :93(<listcomp>)
   123850    1.535    0.000   16.454    0.000 treebank.py:59(
      tokenize)
  1042260    0.982    0.000   15.403    0.000 perceptron.py
     :429(fit_one)
...
```

2.4 Timing with `timeit`

- The `timeit` module allows the developer to compare the relative performance of extensionally equivalent snippets of code for the purposes of optimization. It can be employed in a script, but more often, it is invoked via the command-line or using IPython magic (`%timeit ...`)

```
$ python -m timeit '"-".join(str(n) for n in range(100))'
10000 loops, best of 3: 27.2 usec per loop
$ python -m timeit '"-".join([str(n) for n in range(100)])'
10000 loops, best of 3: 22.3 usec per loop
```

```
$ python -m timeit '"-".join(map(str, range(100)))'
100000 loops, best of 3: 15.8 usec per loop
```

- By default, `timeit` reports the best execution time over three iterations. The idea behind this is that faster-than-average iterations are probably close to the Platonic ideal runtime, whereas slower-than-average iterations usually are due to external factors like overall system load.
- When invoked via the command-line interface (or as `%timeit` in IPython), `timeit` automatically determines the number of loops per iteration by increasing the number of loops by a power of ten until at least 200ms of "wall clock" time has elapsed.