Developer Tools ¶

- Debugging
  - `pdb` module
  - `ipython`
- Timing
  - `time.time()`
  - `timeit` module
- Profiling
  - `cProfile`

Python Debugger `pdb`

Interactively debugging is a great feature of Python. It allows you to step through your program where ever you like and solve issues.

Invoking is as easy as:

```
import pdb; pdb.set_trace()
```

The one line with semicolon isn't great coding practice, but since it's just a temporary thing it's nice to hvae just one line to delete when it's cleared up!

Example: `pdb`

In [ ]:

```
str1 = "Red"

import pdb; pdb.set_trace()

str2 = "White"
str3 = "Blue"

together = ",".join([str1, str2, str3])
print together
```

Combining with Exceptions

```
In [ ]: import traceback

        numerator = 100
        denominators = [1, 8, 0, 3, 2, 12]

        for denom in denominators:
            try:
                answer = numerator / denom
            except ZeroDivisionError as zde:
                full_stack_trace = traceback.format_exc()  # <-- get the full tr
        ace!
                print full_stack_trace
                import pdb; pdb.set_trace()                # <-- drop into a deb
        ugging shell
            else:
                print "We made it! Answer =", answer
```

`pdb` Commands Reference

- `c`: continue
- `p`: print
- `!`: escapes `pdb` and lets you print, say a variable `p` like so: `p !p`

A great article (https://pythonconquerstheuniverse.wordpress.com/2009/09/10/debugging-in-python/) about debugging in Python, and another one (http://ericholscher.com/blog/2008/aug/30/using-pdb-python-debugger-django-debugging-series-/) here.

Timing with `time()` and `sleep()`

Often we'd like to know what is taking so long, or perhaps even wait for a certain period of time.

Unfortunately, this isn't super accurate, and it also gets tedious.

```
In [15]: import time

         # simple timing
         starttime = time.time()
         print "Time elapsed:", time.time() - starttime

         # introduce a wait
         starttime = time.time()
         time.sleep(1)
         print "Time elapsed:", time.time() - starttime
```

```
Time elapsed: 7.20024108887e-05
Time elapsed: 1.00152683258
```

The `timeit` Module

There are actually a lot of complexities to doing truly accurate profiling and timing. You need:

- Accurate time measurement (taking into account OS interrupts, etc)
- Isolation of setup and testing differences between cases
- Simple interface

This module addresses those concerns.

On the Command line

```
$ python -m timeit -n 100000 "1 + 2"
100000 loops, best of 3: 0.017 usec per loop
```

Or, if we need some kind of setup:

```
$ python -m timeit -n 10000 -s "x = range(10000)" "sum(x)"
10000 loops, best of 3: 60.8 usec per loop
```

This is the same as:

In [1]:

```
x = range(1000)
sum(x)
```

Out[1]:

```
499500
```

In Python scripts

Note that of course you can import whatever custom modules you like in the `setup` argument.

In [3]:

```
import timeit
timeit.timeit(setup="a=1;b=1", stmt="a/b") # no error checking
```

Out[3]:

```
0.06045079231262207
```

Example: Try/Except vs. If/Else

The ultimate showdown.

In [1]:

```
import timeit

nonzero_tc = timeit.timeit(setup="a=1;b=1", number=1000000, stmt="try:\n
 a/b\nexcept ZeroDivisionError:\n pass")
zero_tc    = timeit.timeit(setup="a=1;b=0", number=1000000, stmt="try:\n
 a/b\nexcept ZeroDivisionError:\n pass")
nonzero_if = timeit.timeit(setup="a=1;b=1", number=1000000, stmt="if b!=
0:\n a/b")
zero_if    = timeit.timeit(setup="a=1;b=0", number=1000000, stmt="if b!=
0:\n a/b")

print "TC: nonzero=%.6f, zero=%.6f" % (nonzero_tc, zero_tc)
print "IF: nonzero=%.6f, zero=%.6f" % (nonzero_if, zero_if)
```

```
TC: nonzero=0.061591, zero=0.644004
IF: nonzero=0.070239, zero=0.032795
```

`cProfile`: Profiling in Python

Allows profiling automatically an entire Python project or script. Why would you need this over `timeit`?

- Ability to find bottlenecks as opposed to optimizing ones you already know about
- See which functions calls to others are most expensive
- Quick, painless setup without inserting timing code everywhere

Getting the Most out of `cProfile`

`cProfile` is a bit rough on the edges, and newer libraries / wrappers that simplify using it are usually the best bet.

One excellent example is `cprofilev` (https://github.com/ymichael/cprofilev). It allows you to very simply run your programs and get results in an interactive browser interface.

```
$ python -m cprofilev -p 4001 /path/to/python/program.py
```

Then you simply visit http://localhost:4001 (http://localhost:4001).

Interpreting the Output

You'll find an excellent writeup from the author of `cprofilev` here (https://ymichael.com/2014/03/08/profiling-python-with-cprofile.html), but briefly:

Using a Line Profiler

Sometimes you'd also just like to see, line by line, what is causing your problems. Well, you're in luck!

```
    $ pip install line_profiler
```

Then put use the decorator in your script:

```
@profile
def my_func(arg1, arg2):
    # ...
    pass
```

And run with:

```
$ kernprof.py -l -v /path/to/your/script.py
```

The Output

```
Wrote profile results to profiling.py.lprof
Timer unit: 1e-06 s

File: profiling.py
Function: primes at line 2
Total time: 0.00019 s

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     2                                           @profile
     3                                           def primes(n):
     4         1            2      2.0      1.1      if n==2:
     5                                                   return [2]
     6         1            1      1.0      0.5      elif n<2:
     7                                                   return []
     8         1            4      4.0      2.1      s=range(3,n+1
...
...
```

Lab: Profiling

Complete the profiling lab in `excercises/profiling.py`.

You should see directions at the top of that file.

Wrap-up

- Debugging
    - `pdb` module
    - `ipython`
- Timing
    - `time.time()`
    - `timeit` module
- Profiling
    - `cProfile`
    - `line_profiler`