Tools and Principles for the DSI

Lecture Objectives

Understand the following:

1. Development Tools
2. Clean Code
3. Efficient Code
4. Pair Programming

Development Tools

Terminal (iTerm)

- command line (i.e. text-based) way of interacting with system
- useful for
  - git commands
  - installing python packages
  - running python scripts
  - launching Sublime (using “subl” command) or IPython
- iTerm is a nicer version of Terminal

IPython

- It’s a more feature-rich substitute for the Python shell

IPython - View Docstring

View the docstring for any python object using ?object, this is a shortcut for help(obj).

```
?In [5]: ?map
Type:        builtin_function_or_method
String form: <built-in function map>
Namespace:   Python builtin
Docstring:
map(function, sequence[, sequence, ...])  -> list

Return a list of the results of applying the function to the items of...
```

IPython - tab completion

```
In [8]: ma #Press tab...
%macro        %magic        %man          %matplotlib   map           max
```

IPython - access previous results

Use _ for a variable containing the result of the last executed command:

```
In [1]: 9 * 54
Out[1]: 486

In [2]: _
Out[2]: 486

In [3]: x = _

In [4]: x
Out[4]: 486
```

IPython Notebooks

- A feature-rich notebook for Python based on IPython
- Great for
  - prototyping
  - demonstration
  - exploratory data analysis (EDA)
- Terrible for
  - robustly engineered code
- Conclusion:
  - only use notebooks for simple exploratory tasks
  - use Sublime Text for actual development

Sublime Text IDE

- IDE - “integrated development environment”
- IDE integrates:
  - code editing
  - build automation (i.e. running the code)

Git

- Source control
  - Tracks changes to code
  - Allows reverting to previous version of code
  - Synchronizes codebase across many locations
- Widely used in industry
- Other similar tools are TFS (Microsoft) and Mercurial
- Source control is necessary for data science
  - Allows reproducible research
  - Aids collaboration
  - Prevents lost work
- “ABC” - “Always Be Committing”

GitHub

- A website which hosts git repositories
- GitHub and git are not the same
  (Instagram and jpegs are not the same)
- You should sync your local git repositories with your personal GitHub account
- GitHub will host your final project
- Prospective employers look at GitHub repos

Style and Structure

```
t=1*10**-10**2
m=1*10**2
def f(f, f1, q, t, m):
    i = 0
    while ((f(q) > t)
        and
        (i < m)):
        i,q=i+1,q-f(float(q))/f1(q)
    return q

f2 = lambda x:x**2
f3 = lambda x:2*x

print( f(f2, f3, 10, t, m)
       )
```

```
def find_zerc(f, f_prime, x,
              threshol=1E-100, max_iter=1E100):
    """
    Finds the zero of a function f, given its derivative
    function f_prime, using the Newton-Rapshon method:
    https://en.wikipedia.org/wiki/Newton%27s_method
    """
    x = float(x)
    iterations = 0
    while f(x) > threshold and iterations < max_iter:
        iterations += 1
        x = x - f(x)/f_prime(x)
    return x

if __name__ == '__main__':
    def f(x): return x**2

    def f_prime(x): return 2*x

    initial_guess = 10

    print "The solution is: %s' % find_zerc(f, f_prime,
                                            initial_gues);
```

Style and Structure

- Code is read more than it is written; style *is* substance
- Structure your code into functions, modules, and classes
- Follow the DRY principle:
  - DRY = “don’t repeat yourself”
  - in contrast to WET = “we enjoy typing”
- Read this, a lot: http://legacy.python.org/dev/peps/pep-0008/
- Also this: https://google-styleguide.googlecode.com/svn/trunk/pyguide.html

Writing Efficient Code

- Code that analyzes a lot of data can take forever to complete
- Optimizing your code can be the difference between code that takes a few minutes to run and code that will effectively never finish running
- “Runtime analysis” is a very popular interview topic

Writing Efficient Code

```
1   def find_anagrams(lst):
2       result = []
3       for word1 in lst:
4           for word2 in lst:
5               if word1 != word2 and sorted(word1) == sorted(word2):
6                   if word1 not in result:
7                       result.append(word1)
8                   if word2 not in result:
9                       result.append(word2)
10       return result
```

How many comparisons does this code perform?

Assume N words in the input list, K of which will go in the outcome list.
N*N comparisons between each item in the input, plus >K*(K-1) total comparisons for lines 6 and 8.

Writing Efficient Code

```
1  def find_anagrams(lst):
2      result = []
3      d = defaultdict(list)
4      for word in lst:
5          d[tuple(sorted(word))].append(word)
6      for key, value in d.iteritems():
7          if len(value) > 1:
8              result.extend(value)
9      return result
```

How many comparisons does this code perform?

Assuming N words in the input list, then there are N lookups in the dictionary to fill it with the input, and a loop through <N items to determine the output.

How?

Hashing!

Dicts and Hash Tables

- A hash table is a data structure that maps keys to values

```
homestate = {"giovanna": "maine", "ryan": "california",
        "katie": "michigan", "zack": "new york"}
```

- Python dictionaries are an implementation of hash tables
- Instead of iterating through a list of tuples:

```
("giovanna", "maine"), ("ryan", "california"),
("katie", "michigan"), ("zack", "new york")]
```

  You can access a key's value directly:

```
homestate['katie']
```

- Key takeaway: use dict (and similar structures) instead of list to organize your data

Mutability

- Mutable objects can change their value but keep their `id()`
  - lists
  - sets
  - dictionaries
- Immutable objects cannot be altered. A new object has to be created if a different value has to be stored.
  - They play an important role in places where a constant hash value is needed, for example as a key in a dictionary.
- Key point: only immutable types are hashable, so only immutable types can be used in sets or as dictionary keys

Dict operations

Looping

```
GOOD: for key in dictionary:
BAD:  for key in dictionary.keys():
GOOD: for key, value in dictionary.iteritems():
BAD:  for key, value in dictionary.items()
```

Equality

```
dict1 == dict2
```

Checking Membership

```
GOOD: key in dictionary
BAD:  key in dictionary.keys()
```

Set

- A set is like a dictionary with only keys and no values
- Sets are useful for checking membership and deduplication. For example:
  - `n in my_list` takes `len(my_list)` steps
  - `n in my_set` takes 1 step
- Example: get all the unique words in a string that are longer than 3 characters:

```
s = set()
for word in string.split():
    if len(word) > 3:
        s.add(word)
```

- Sets are also useful for removing duplicates in a list (if you don't care about order):

```
L_unique = list(set(L))
```

Itertools

Combinatoric generators:

Iterator	Arguments	Results
`product()`	p, q, ... [repeat=1]	cartesian product, equivalent to a nested for-loop
`permutations()`	p[, r]	r-length tuples, all possible orderings, no repeated elements
`combinations()`	p, r	r-length tuples, in sorted order, no repeated elements
`combinations_with_replacement()`	p, r	r-length tuples, in sorted order, with repeated elements
`product('ABCD', repeat=2)`		AA AB AC AD BA BB BC BD CA CB CC CD DA DB DC DD
`permutations('ABCD', 2)`		AB AC AD BA BC BD CA CB CD DA DB DC
`combinations('ABCD', 2)`		AB AC AD BC BD CD
`combinations_with_replacement('ABCD', 2)`		AA AB AC AD BB BC BD CC CD DD

From: https://docs.python.org/2/library/itertools.html

Lambda Functions

```
def add(x,y):
    z = x + y
    return z
```

```
lambda x,y:x+y
```

Generators

- The `range` function will create a list of length `n` and then we iterate over it.
  - ```
    for i in range(n):
        print i
    ```
  - This wastes memory by creating a list of length `n`
- We can use a *generator* to save the memory:
  - ```
    for i in xrange(n):
        print i
    ```
  - The `xrange` function will generate the next value when it's needed, but won't return an entire list like the `range` function.

Looping Tools

- Simplest, most Pythonic loop
  - ```
    for item in L:
        print item
    ```
- Use `enumerate` (a generator) when you need the index too
  - ```
    for i, item in enumerate(L):
        print i, item
    ```
  - ```
    for i in xrange(len(L)):
        print i, L[i]
    ```
- Use `zip` or `itertools.izip` (a generator) to combine two lists:

```
first_names = ['Giovanna', 'Ryan', 'Jon']
last_names = ['Thron', 'Orban', 'Dinu']
In [3]: zip(first_names, last_names)
Out[3]: [('Giovanna', 'Thron'), ('Ryan', 'Orban'), ('Jon', 'Dinu')]
```
```

Pair Programming

What is it?

- One computer, 2 keyboards
- Driver + Navigator

Pair Programming

Why?

- Learn more
- Higher quality output
- Good practice
  - collaborating and communicating at length about complex problem
  - working with different skill sets and personalities
- Increasingly popular in industry

Pair Programming Core Principles

- Get to know your partner: “What did you think about the lecture?”
- Take turns: trade driver and navigator roles every 30 minutes
- Listen: if your partner asks “Why are we doing that?” then take the time to answer.
- Be patient: if your partner types something that looks wrong, try to understand it before correcting it.
- Be clear: explaining technical concepts is hard. Practice.
- Be humble
- Disagree productively
- Switch partners daily