Computational Frameworks for Social Science

Matt Dickenson*

Current as of: February 1, 2012

1 Introduction

Table 1: The Software World According to Josh Cutler

Software Engineers	Computer Scientists	Hackers/Coders
IBM	Google or PhD	Start-ups
Wear Ties	Beards	Build Apps

1.1 Source Control

We will use `Git` as our source control. `Git` was a sea change in CS. Git can save versions A and $A+1$ simultaneously, so that you can work with multiple versoins of code and cooperate with other people. To use:

```
git init
git add text.txt
commit -m ‘‘first commit’’
```

Once we commit, this flushes the change set and creates "first commit" in the git log.

```
git add text.text
commit -m ‘‘after edits’’
```

*Notes for Josh Cutler's PS 398 course, Duke University, Spring 2012.

To fully preserve for posterity:

```
git push .
```

To create a new version of yourself:

```
git pull .
```

The key idea here is that you can push to/pull from arbitrary places. There is no hierarchy for what is the True version.

1.1.1 A Word on Branching

Branching is how code gets built. Branching is a way to expand a codebase while keeping an earlier version stable. When someone talks about the "hot new α version", they are talking about a branch. For instance, we may push a package to CRAN (v1.0), then start work on an update (v2.0). In the meantime, we find out a bug in v1.0 that we need to fix. With branching, this isn't a problem.

1.1.2 Naming Versions

Consider a version $X.Y.Z$. We commit $X.Y.Z + 1$ when we have fixed a bug. We commit $X.Y + 1.Z$ when we have added a feature. We commit $X + 1.Y.Z$ when we want to charge people more money.

1.1.3 Commit History

A basic rule of thumb is: if you can't describe what you did in a single line, it's too big a jump.

Sometimes in professional software you will see "commit wars" where a couple of people just push the same junk back and forth until someone fires them. For example, one guy may indent his Python code with two spaces, while another uses four. Agree on this early. Communicate.

1.2 Testing

Automated testing, that is. We're going to use code that we write to test our other code. Specifically, we're going to use "unit tests" (as opposed to "integration testing"). Testing

is contentious–people will fight about, leave companies over, or start companies because of this.

A motivating example is BDD vs. TDD–behavioral driven design *versus* test driven design. BDD people are masturbatory and annoying. TDD people write tests first and then code. They write a series of tests that suitable code should pass, and then design code. Others write code and then test it, but they think of the same test cases they did when they were coding, so they may overlook problems. Your choice will likely depend on how long you are going to have to maintain the code. Technically, TDD code has to fail a test before there is a bug.

But we are talking at a non-philosophical level about unit tests. Reasons to use TDD:

- if someone else inherits your code, it helps them to have a test suite to detect hairy portions of code
- it helps readers to see what the code is actually supposed to do
- you can refactor (rewrite) your code with more confidence, since you can quickly check all known relevant tests (without TDD, you will undoubtedly introduce bugs; with TDD you will be at least as good as you were before)
- it helps you be lazy–most people don't have enough mental RAM to remember everything a program does
- you will write better code if you know you have to test it
- if you truly understand what code is supposed to do, you should be able to assert what output it should give you

Python is dynamic (duck) code. The important thing to remember is that there are more bugs that can be introduced than, for example, C++. To see what we're talking about, let's actually write some code.

The two things that we check for are *correctness* and *robustness*. Correctness asks, "does the code do what we want it to when we give it the right input?" Robustness asks, "how does the code handle unexpected inputs?" In Python's `unittest` library, a test can have three outcomes: "." (pass), "F" (fail), or "E" (error). The key point: have a failure plan. This is important, for example, when writing a web scraper. HTML is supposed to look a certain way, but in reality it never does.

The process is iterative:

1. see a bug
2. write a test
3. fix the code so that it satisfies the test
4. repeat

Again, the argument for using code is that your mind will be more open (and thus write more appropriate tests) before you develop the tunnel vision that comes with writing code in a certain way.

2 Object-Oriented Programming

2.1 Mini-Homework

Restructure homeworks on github so that the root looks like `/HW1` and use `/HW2` for the next one.

2.2 What is OOP?

Table 2: Three Styles of Programming

Procedural	Functional	Object-Oriented (OOP)
Probably what you think of	You don't care about this	Philosophy of representation
C, R		Python
Sufficient for small projects		
Very little hierarchy or structure	list comprehensions, etc.	tightly couples data w/methods

Consider an object `Book`. In `R`, this would have a bunch of characteristics like "author" and "pages". In OOP, every book that exists is an instance of the class "books." In fact, *everything is an object* in Python: numbers, strings, classes. Think of classes as a way to give your object some structure.

2.3 Why OOP?

To stay DRY: Don't Repeat Yourself. Ultimately, you will be writing less code using OOP– even though it won't seem that way at first. A lot of it comes down to being able to say, "This is a `thing`. It should have `this` info and `these` methods." Any code can be written procedurally *or* as OOP, but the choice is which is simpler.

2.4 Inheritance

Python is a method-passing language, which means it does something called "dynamic dispatch." Say we defined:

```
class Animal(object):
  def __init__(self,name):
```

```
    self.name = name

class Cat(Animal):
  def talk(self):
    return ‘‘Meow!’’
```

So when if we set `a` equal to `Animal(‘‘Fido’’)` and `b` equal to `Cat(‘‘Sally’’)`, then when we call `b.talk()`, it will search within class Cat first, and then class Animal if it doesn't find anything.

This is known as *polymorphism*, but it won't mean much to you until you practice.

As the example `sports.py` will show, there is a trade-off between generality and specificity in how much you want a subclass to inherit from a superior class. Practicing this will force you to think about:

- What should exist at the object level?
- How should those things interact with each other?

Pro-tip: Move as far away from `global` variables as possible.

3 Pragmatic Programming

3.1 Data types

There are three data types in Python, shown in Table 3.

Table 3: Data Types

Name	Code	Details
tuple	`()`, `a=(1,2)`	immutable, i.e. you can't change it
list	`[]`, `a=[], a[0] = ''foo'', a[1]=''bar''`	
dictionary	{}, `a[''foo'']=''bar''`	use of keys

3.2 Exceptions

Any time Python "explodes" it is because an exception has been raised. `Exception` is a class, and all types of exceptions inherit from this class. Any exception based on the class `Exception` will have the attributes `msg` (what prints when the exception is raised) and `stack trace` (which identifies where the error occurred). Note that "raise" is a technical term in Python.

```
class CustomException(Exception): # inherits from Exception
  def __init__(self, value):
    self.value = value

  def __str__(self):
    return self.value

def i_call_a_function_with_errors():
  try:
    print "Calling a function...."
    #function_with_generic_error()
    #function_with_custom_error()
    #function_with_unknown_error(1)
    print "Tada!"
  except CustomException as inst: # 'as' gives us access to the exception
```

```
    print "Custom Error Caught! Error({0})".format(inst.value)
  except: # any exception is caught, even ones you don't know about
    print "Default Error Caught!"
  else: # if nothing broke, then run this block
    print "No error raised."
    traceback.print_exc() # this prints the traceback
  finally: # this block is always run
    print "Goodbye!"

def function_with_generic_error():
  raise Exception, "Foo!" # this method doesn't know what to do with
the exception

def function_with_custom_error():
  raise CustomException, "Foo Bar!" # this will be handled in the
function above}

def function_with_unknown_error(foo):
  foo.bar()
```

Caught exceptions are ones that keep the user from doing something the programmer didn't want them to do. Uncaught exceptions cause problems.

In a Python test suite, you set up a `try` block to run some code where you think a problem might happen. As soon as an exception is thrown in your `try` block, it doesn't try anything else.

Rule of thumb: If you know what to do with an error, handle it. If you don't, let it percolate up.

Exceptions are something that you've been dealing with up to now but (maybe) didn't even know about.

3.3 Algorithms

Having an instict for better and worse algorithms will let you know whether solving your problem will take hours or years.[1] An algorithm can be defined as "a series of steps that

[1] "No one in stats gets a Ph.D. without coding and the only way people get Ph.D.'s without coding in our discipline is because there are some dumb people who have made their way up, but those guys are going

achieve a desired outcome."

Consider the task of sorting a list. This is not a problem you will ever have to solve because better people have already done it, but it's easy to visualize. We use algorithms to solve hard problems. In mathematics it's known as an NP problem. (This is just nerd's way of saying "hard.") Most problems in game theory are NP complete.

In political science, you might be using a data set with 18 million events. But the thing you're interested in might require $18,000,000^2$ or even $18,000,000^3$ observations. 18 million cubed is a big number.

To put this in a Python context, say we want to sort a list $[y_i, ...y_n]$. A terribly inefficient way to do this would be to randomly shuffle and then check to see if they're in order. There are $n!$ ways to shuffle the list, so the problem grows factorially.

In programming, we use "big O" notation to talk about complexity. The O means $\forall x f(x) < cf(x)$. So if we have a problem $n^2 + n$, we denote it $O(n^2)$ and call it "quadratic complexity." Complexity of the class $O(n)$ is called "linear," and so on. It tells us how the number of operations goes up as n grows. We use it to speak about *average* complexity of problems–after all, we can only speak in averages.

3.3.1 Selection sort

```
k = 0, L= [] \\
Loop through n-k \\
    find smallest number at j \\
    swap L[k] with L[j] \\
    k++
```

```
              [5,1,15,7,111]  # of things to check
Rd 1: k=0,j=1 [1,5,15,7,111]    5
Rd 2: k=1,j=1 [1,5,15,7,111]    4
Rd 3: k=2,j=1 [1,5,7,15,111]    3...
Rd 4: k=3,j=3
```

to die soon." – Scott DeMarchi.

3.3.2 A Brief Primer/Refresher in Discrete Math

$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$, which is a $O(n^2)$ (quadratic) complex problem. How do you prove it?

$$\begin{aligned}
\sum_{i=1}^{n} i &= n + (n-1) + (n-2) + ... + 3 + 2 + 1 \\
2 \times \sum_{i=1}^{n} i &= n + (n-1) + ... + 2 + 1 + 1 + 2 + ... + (n-1) + (n-2) \\
&= (n+1) + (n+1) + ... \\
&= n(n+1)
\end{aligned}$$

Go back to the example above. How many tries would the best case take? 5. The worst case? 5. And the average case? You guessed it–5.

3.3.3 Merge Sort

This method would take the list to be sort it, split it in half again and again until they were all disaggregated to the unit level. It would sort those and reassemble (merge) them into a two-ple (get it?). This reduces the list sort to an $O(nlog(n))$ problem. (In this case we're not even throwing away little numbers or constants from the n.)

Try this out. Come up with a list of numbers, maintain a spot in them. It's fairly easy to sort two things and interweave them. It will take $nlog_2(n)$ sorts.

3.3.4 What should I care about in optimizing an algorithm?

Some people will care about the time it takes. Others will care about the (memory) space it take. Most people who care about memory work on rocket ships or microwaves. Merge sort takes more space than selection sort, but uses less time. As with anything in life, it's a tradeoff.

3.3.5 Back to NP

P means that the problem is in polynomial time class–not that your algorithm is in P time (it may be exponential) but that the ideal answer is in P time. Computational game theorists sometimes look at *whether* something is solvable in polynomial time. If it isn't, we call it NP time–that is, not solvable. If you can prove that $P = NP$, you can break all cryptography in

Table 4: How long will it take to find y_i?

Method	List	Worst	Best	Average
Naive	$[y_1, ..., y_n]$	$O(n)$	$O(1)$	$O(\frac{n}{2})$
Binary	sorted	O(log n)		

the world, make a lot of money, and retire. If somebody says a problem is "NP," that means don't waste your time. (Actually they'll say "NP-hard," "NP-complete" and so on–that just means someone way more mathematically inclined than us has proven it so; don't argue.)

3.3.6 Quick Sort

There is a fourth sort, which we will not get into here, that most programs actually use. Quick sort is, on average $O(nlogn)$, but its worst case is n^2. Again, it all comes back to what you care about this.

Don't stress too much about actually computing complexity classes–they've already been computed for almost anything interesting enough that you'd want to work on it. Just know how to choose between them when presented with options.

Remember that there is a trade-off between how much time you spend programming the algorithm and how much time it will save you. In general, think about how many times you will plausibly be running the algorithm. Most working programmers look for "satisficing" solutions–fast enough and no faster.

Homework:

1. generate arrays of numbers
2. sort them
3. count the time
4. graph it (N on X axis, time on Y)

Implement the sort algorithms first. Everything else will be trivial.