L08: Review and Practice

Eric Franzosa (franzosa@hsph.harvard.edu)

http://franzosa.net/bst273

Overview

- Last Monday's lecture bumped to today
  - Will skip one of the later "special topics" (modules) lectures to compensate
- HW4 posted (due Friday 12/6)
- Draft Final Project specifications this Weds (12/4)
  - To help with default vs. custom Final Project choice
  - Officially starts next Monday (12/9)
- Comments on patterns from HW3
- Comments on Jupyter vs. command-line (scripted) Python
- Command-line navigation overview
- **Live coding of the scheduling problem from HW1**
- Feedback on live coding

Comments on Patterns from HW3

General objectives for programming

- (1) Arrive at the correct answer in a reasonable amount of time
  - Including the time it takes to write the code!
- (2) Write code that is easy for someone to understand in the future
  - Including future you!
- (3) Write code in a way that is less likely to introduce errors, now or later
  - Practice defensive coding
- *One of the easiest ways to achieve 2 and 3 is to avoid writing code that is repetitive and/or hard to read (e.g. deeply nested code).*

Avoid writing repetitive code: vowel counting

vowels1.py

```
def count_vowels( text, lower=False ):
    ret = {}
    for char in text:
        if char == “A”:
            ret[“A”] = ret.get( “A” ) + 1
        elif char == “E”:
            ret[“E”] = ret.get( “E” ) + 1
        ...
        elif lower and char == “a”:
            ret[“A”] = ret.get( “A” ) + 1
        elif lower and char == “e”:
            ret[“E”] = ret.get( “E” ) + 1
         ...
    return ret
```

- This *would* return the right answer in a reasonable amount of time!
- Lots of repetition
- Not fun to read or type
- *Note: I recoded this myself, it isn’t based on any one person’s solution*

Avoid writing repetitive code: vowel counting

vowels1b.py

```
def count_vowels( text, lower=False ):
    ret = {}
    for char in text:
        if char == “A”:
            ret[“A”] = ret.get( “A” ) + 1
        elif char == “E”:
            ret[“E”] = ret.get( “E” ) + 1
        ...
        elif lower and char == “a”:
            ret[“A”] = ret.get( “A” ) + 1
        elif lower and char == “e”:
            ret[“E”] = ret.get( “A” ) + 1
        ...
    return ret
```

- Very easy to make a copy-and-paste mistake in one of the blocks that is hard to catch

Avoid writing repetitive code: vowel counting

vowels1c.py

```
def count_vowels( text, lower=False ):
    ret = {}
    for char in text:
        if char == “Y”:
            ret[“Y”] = ret.get( “Y” ) + 1
        elif char == “A”:
            ret[“A”] = ret.get( “A” ) + 1
        elif char == “E”:
            ret[“E”] = ret.get( “E” ) + 1
        ...
        elif lower and char == “a”:
            ret[“A”] = ret.get( “A” ) + 1
        elif lower and char == “e”:
            ret[“E”] = ret.get( “E” ) + 1
         ...
    return ret
```

- Easy to introduce errors later
- Ex: Updating to count “Y” as a vowel
  - We remembered to update the uppercase condition…
  - …but not the lowercase condition

Avoid writing repetitive code: vowel counting

vowels2.py

```
def count_vowels( text, lower=False ):
    ret = {}
    if lower:
        text = text.upper( )
    for char in text:
        if char in “AEIOU”:
            ret[char] = ret.get(char,0) + 1
    return ret
```

- If we’re treating upper and lowercase the same, just uppercase the whole text before counting
- Replace many small tests with one larger sophisticated test

Avoid writing deeply nested code (when possible)

primes1.py

```
def prime_range( n ):
    primes = []
    for n2 in range( 2, n ):
        # check if n2 is prime
        IS_PRIME = True
        for divisor in range( 2, n2 ):
            if n2 % divisor == 0:
                IS_PRIME = False
        if IS_PRIME:
            primes.append( n2 )
    return primes
```

- This is an alternate prime range function (more nested, harder to read, easier to break)

Avoid writing deeply nested code (when possible)

primes2.py

```
def check_prime( n ):
    IS_PRIME = True
    for divisor in range( 2, n ):
        if n % divisor == 0:
            IS_PRIME = False
    return IS_PRIME

def prime_range( n ):
    primes = []
    for n2 in range( 2, n ):
        # check if n2 is prime
        if check_prime( n2 ):
            primes.append( n2 )
    return primes
```

- The way that we actually composed this code is simpler: take a repeated action out of `prime_range()` and implement it as it's own function, `check_prime()`.
- Once we are confident that `check_prime()` is working, any errors we find must be due to problems with `prime_range( )`.

Repetitive and deeply-nested code: weird sums

weirdsum1.py

```
def weird_sum( numbers ):
    ret = 0
    for n in numbers:
        if n < 0:
            n = -1 * n
            if n % 2 == 0:
                n = n // 2
            elif n % 2 == 1:
                n = 2 * 2
        else:
            if n % 2 == 0:
                n = n // 2
            elif n % 2 == 1:
                n = 2 * 2
        ret = ret + n
    return ret
```

- Rules, given some numbers:
  - Sum all numbers
  - If number < 0, add its absolute value
  - If number is even, add half its value
  - If number is odd, add twice its value
- This *would* return the right answer in a reasonable amount of time!
- Hard to read, *especially* from the many levels of nesting
- *Note: I recoded this myself, it isn't based on any one person's solution*

Repetitive and deeply-nested code: weird sums

weirdsum2.py

```
def weird_sum( numbers ):
    ret = 0
    for n in numbers:
        if n < 0:
            n = -1 * n
        if n % 2 == 0:
            n = n // 2
        elif n % 2 == 1:
            n = 2 * 2
        ret = ret + n
    return ret
```

- No need for deep nesting: can apply the transformations serially

Jupyter vs. Command-line Python

Jupyter vs. command-line Python

- **Jupyter** is good at…
  - Experimenting with small chunks of code
  - Interactive analysis
  - HW1, HW2, HW3
- **Jupyter** is not great at…
  - Organizing and tracking state
  - Running as an automated program
  - Interacting with command-line (user) options
- **Command-line Python** is good at…
  - Behaving like a traditional computer program
  - Operating as part of an analysis workflow
  - HW4, Final Project

Command-line Python runs in one go, top-to-bottom

code.py (*open in Atom*)

```
from math import sqrt
```
1

```
x = 16
```
2

```
print( sqrt( x ) )
```
3

```
def squared( x ):
    return x ** 2
```
4

```
x = 5
```
5

```
x = 2 * x
print( squared( x ) )
```
6

(*a terminal*)

```
$ python code.py

  4
  100
```

Command-line Python stops at first error

code.py (*open in Atom*)

```
from math import sqrt
```
1

```
x = 16
```
2

```
print( sqrt( y ) )
```
3

```
def squared( x ):
    return x ** 2
```

```
x = 5
```

```
x = 2 * x
print( squared( x ) )
```

(*a terminal*)

```
$ python code.py

  NameError: name 'y' is not defined
```

Jupyter runs as we please

code.ipynb (*Jupyter input cells*)

```
from math import sqrt
```

```
x = 16
```

1

```
print( sqrt( x ) )
```

```
def squared( x ):
    return x ** 2
```

```
x = 5
```

```
x = 2 * x
print( squared( x ) )
```

2

Jupyter output

```
NameError: name ‘squared' is not defined
```

Jupyter runs as we please

code.ipynb (*Jupyter input cells*)

```
from math import sqrt
```

```
x = 16
```
1

```
print( sqrt( x ) )
```

```
def squared( x ):
    return x ** 2
```
3

```
x = 5
```

```
x = 2 * x
print( squared( x ) )
```
4

Jupyter output

```
1024
```

Jupyter runs as we please

code.ipynb (*Jupyter input cells*)

```
from math import sqrt
```
6

```
x = 16
```
1

```
print( sqrt( x ) )
```
7

```
2.2360679775
```

```
def squared( x ):
    return x ** 2
```
3

```
x = 5
```
5

```
x = 2 * x
print( squared( x ) )
```
4

Jupyter output

```
1024
```

Refresher on running Python from the command line

Refresher on navigating the command line

Refresher on command-line navigation

- All of this is explained in much more detail in Module 0
  - With practice, if interested
- Here's the minimum you'll want going forward

Command-line task…	Windows	MacOS / Linux
View current location/folder	`cd`	`pwd`
View current contents	`dir`	`ls`
Move (down) into subfolder	`cd` *name*	`cd` *name*
Move (up) into parent folder	`cd ..`	`cd ..`

Transition to live-coding exercise

Feedback on live-coding exercise?