Learning to program from your peers

Hunter Blanks, hblanks@artifex.org / @tildehblanks

October 22, 2011

“Your typical computer programming class goes something like this:

> Each time the class meets, the instructor lectures on some new programming concept, and assigns homework. Then the students go home, or to the lab, and try to write programs that do what the assignment wants. Maybe they work; maybe they don't; maybe the students stay up late at night trying to find the bugs in their programs before the deadline. They hand in their programs. Somebody grades them and they get back a number that's supposed to indicate how good their program is. Then it's on to the next lecture and the next assignment.

What's wrong with this picture?

How are students supposed to become better programmers, and in particular programmers who appreciate virtues other than just functionality, with so little feedback?”

> "I'd like the story to look more like this:
>
> > Each time the class meets, it begins with a review of the student work done that week. Maybe there isn't time to go through all of the submissions, but some of the interesting ones have been picked out. The class critiques the work together, asking the author about what they find interesting or unclear in the code. In this way, the whole class learns what makes code easy or hard to understand. Maybe a student got most of the assignment working, but wasn't able to get everything working perfectly. Instead of simply handing the student a low grade, the instructor can bring up the work for the whole class to debug together, until it works, and everyone has learned something. When the students go off to do the assignments, they know their work might be on the big screen next week. So they try hard to make it look good. The students are encouraged to talk about the assignments and work on them together. They don't have to be isolated by strict rules to prevent plagiarism, because they have to be prepared to defend their work to the class. "
>
> –Ka-Ping Yee, "Programming as Design", http://zesty.ca/bc/design.html

Left to right: Tim Peters, Jeremy Hylton, Guido van Rossum, Fred Drake, Ka-Ping Yee. Barry Warsaw is missing from this picture. The picture was taken in July 2000. (http://zesty.ca/bc/pythonlabs.html)

This, then, was the method behind UC Berkeley's spring 2003 class CS98, aka "Beautiful Code". It spoke, and still speaks to at least three problems:

1. Programmers are never taught how to look at or talk about code. (Much like we are only sometimes taught how to look at or talk about paintings)

"That's a beautiful painting."

"As for the table, it is not clear whether it is parallel to the picture plane or at an angle. This question has given rise to conflicting interpretations including the theory of multiple viewpoints. Confusion arises because of the ambiguous dark area in the lower left. Is it the table or is it fabric?'
–Mary Adam, Visual Artist

2. Programmers are rarely taught good design practices.

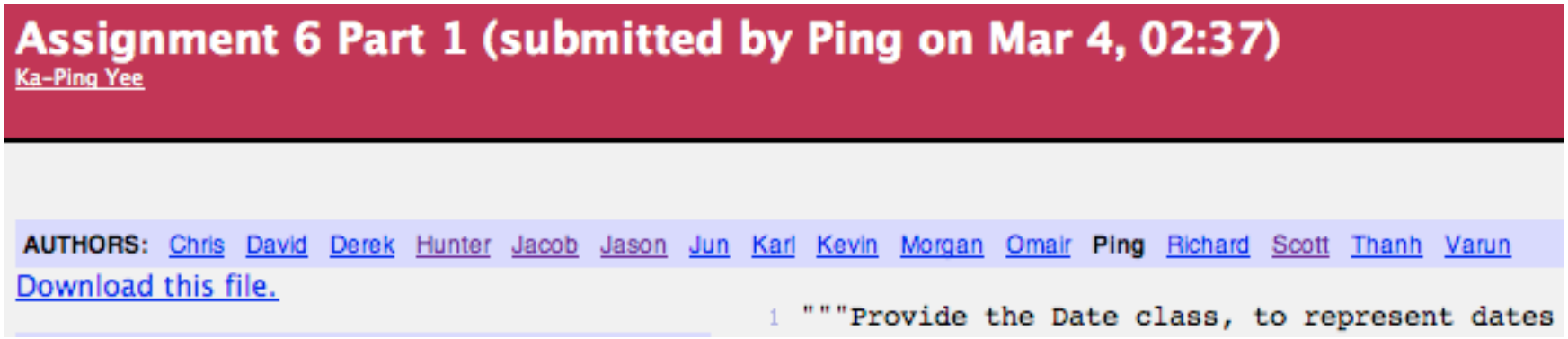
```
>>> import this
The Zen of Python, by Tim Peters

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.
```

Beautiful? Ugly? And what does he mean "readability?"

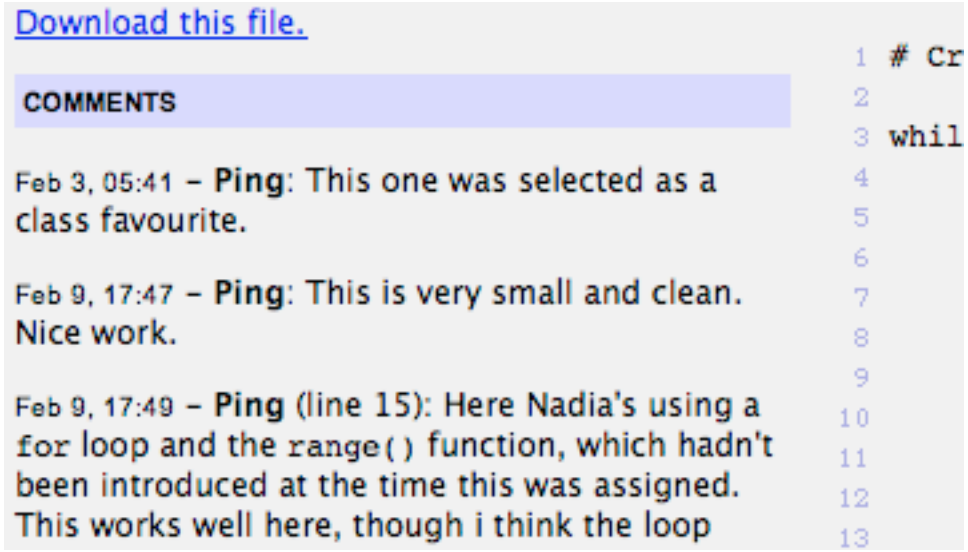
3. Both of these things are best learned by example.

Four highlights from this method are:

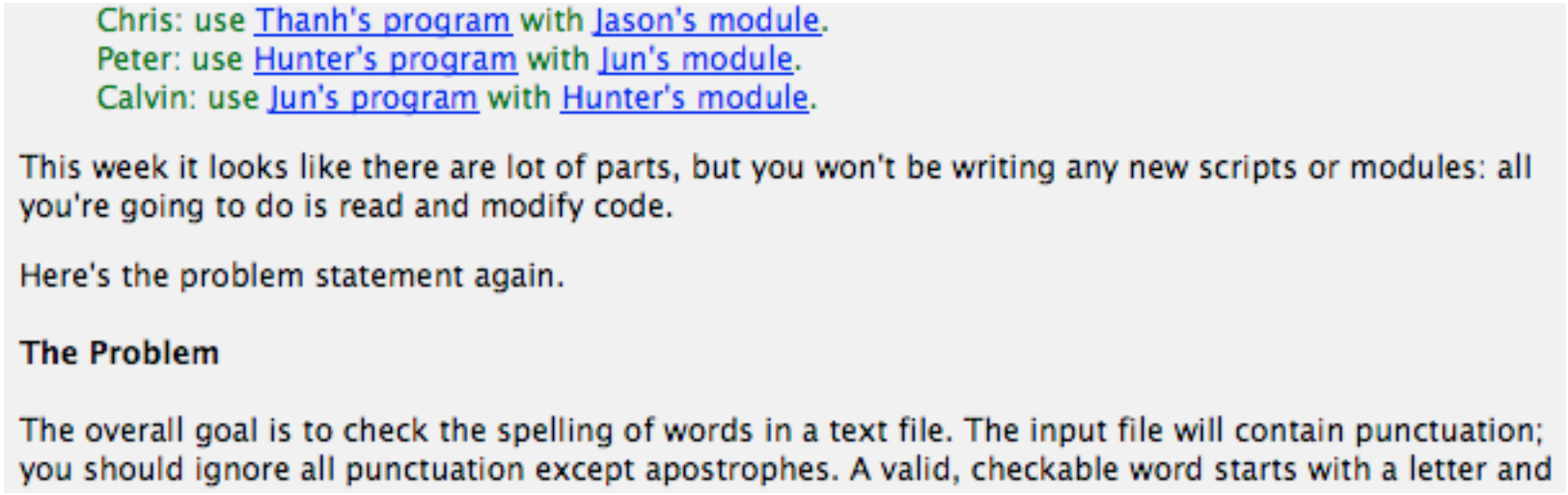
1. After the deadline, all assignments are made visible to everyone.



2. Comments (general, or line by line) are actively sought from everyone, both before and in class.



3. In many assignments, your starting point is the code written by one of your classmate's from a prior assignment. Also, your endpoint may not be an isolated program — it could be a list of function definitions and docstrings (with no code at all!), unit tests, or an integration of other people's modules.



Chris: use Thanh's program with Jason's module.
Peter: use Hunter's program with Jun's module.
Calvin: use Jun's program with Hunter's module.

This week it looks like there are lot of parts, but you won't be writing any new scripts or modules: all you're going to do is read and modify code.

Here's the problem statement again.

The Problem

The overall goal is to check the spelling of words in a text file. The input file will contain punctuation; you should ignore all punctuation except apostrophes. A valid, checkable word starts with a letter and

4. In every class, the bulk of instruction time is dedicated to exploring new parts of the language.

Curiously enough, the opposite of `in` is `not in`.

```
>>> 3 not in [1, 2, 4]
1
>>>
```

Q8. Write a little function that takes a sentence and converts it (very simplistically) into Pig Latin. Each word that begins with a vowel should have "way" appended in the resulting sentence; each word that begins with a consonant should have the consonant moved to the end of the word and "ay" appended after that. Don't worry about punctuation, combined consonants, or capital letters; just handle these two simple cases:

```
>>> def piglatin(sentence):
...     <you fill this in>
...
>>> piglatin('  ethel the aardvark goes quantity surveying.')
'ethelway hetay aardvarkway oesgay uantityqay urveying.say'
>>>
```

When you get here, **find someone you haven't met yet and show them your Pig Latin program.**

Your assignment today (pick one):

1. (Directly from Ka-Ping Yee) Write a program that reads numbers from keyboard input and prints their prime factors. It should behave like this:

```
soda% python factorize.py
Number: 34234
 34234 = 2 * 17117
Number: 938
 938 = 2 * 7 * 67
```

2. (Shorter) Write a function that takes an integer and returns the corresponding number in the Fibonacci sequence. It should behave like this:

```
>>> fib(0)
0
>>> fib(4)
3
```

Thank you!

The sources for this talk are:

zesty.ca/bc/
maryadamart.com/

You can find this talk online at:

github.com/hblanks/talks/

You can reach me at:

hblanks@artifex.org / @tildehblanks
hblanks@monetate.com

(And yes, we **are hiring** at Monetate, for many, may things)