CS61A Lecture 22

Friday, October 18th, 2019

Announcements

- Extra office hours today from 4 to 8pm in the Wozniak Lounge.
- Homework 5 due next Tuesday.
- Midterm 2 covers content through Wednesday, 10/16.
    - No `BTree` class on this midterm.
    - You can bring 2 two-sided sheets of handwritten notes.
    - Different treatment of orders of growth.
    - Monday will be review of concepts and exams.
    - Seats will be assigned next Wednesday.
    - No lecture on 10/23, and no discussion next week.

Decomposition

Today, we will be working with a slightly larger example than usual for a lecture. We will talk through why the 61A Projects are laid out the way they are, to prepare you for 61B projects, in which you will have to design your project layout.

One way to do that is through modular design, in which you want to design your project by modular parts through abstraction barriers, instead of worrying about minor details in some other function when writing code.

Separation of Concerns

A design principle in which you isolate different parts of the program that address different concerns.

This way, you can build modular components can be developed and tested independently. In a perfect world, you can write the components, specify how they fit together, and your program works!

The Hog project was designed this way!

Component: Hog game simulator	**Component: Game commentary**	**Component: Player strategies**
Game rules	Event descriptions	Decision rules
Ordering of events	State tracking to generate commentary	Strategy parameters (e.g. margins and number of dice)
State tracking to determine the winner		

You can see that each component is built independently of each other. The simulator doesn't care how its results are printed, and the game commentary doesn't care how the player plays.

So is the Ants project:

Component: Ants game simulator	**Component: Actions**	**Component: Tunnel structure**
Order of actions	Characteristics of different ants and bees	Entrances and exits
Food tracking		Locations of insects
Game ending conditions		

Same goes for Ants! Let's use an example to see how a program built from scratch.

Restaurant Search Data

Yelp used to release open data sets about restaurants near the Berkeley campus, and there is still some older data which you can find on cs61a.org under the filename 22.zip.

Each piece of data under the business listings looks like this:

```
{"business_id": "gclb3ED6uk6viWlolSb_uA", "name": "Cafe 3", "stars": 2.0, "price": 1, ..
```

It consists of:

- Business ID: A randomly generated, unique ID for each restaurant.
- Name: the name of the restaurant
- Stars: the star rating, out of five
- Price: The dollar sign category, with 1 being the cheapest

And there is also a section for reviews:

```
{"business_id": "gclB3ED6uk6viWlolSb_uA", "user_id": "xVocUszkZtAqCxgWak3xVQ", "stars":
```

Let's build a program that can take a restaurant, and recommend similar ones. What is similar? We'll have to decide what that is.

A lot of files you download from the Internet will come in this format, which looks like Python dictionaries, but is actually in a format called JSON (JavaScript Object Notation). Python can read it though, which is fine.

So let's begin with the interface:

```
results = search('Thai')
for r in results:
    print(r, is similar to', r.similar(3))
```

OK. Now that we know how our program will work, let's build the code to make sure this works.

```
def search(query, ranking=lambda r: -r.stars):
    results = [r for r in Restaurant.all if query in r.name]
    return sorted(results, key=ranking)
```

We haven't built the restaurants class yet, but now we know it will have a name attribute and a class attribute called `all` which is an iterable. And we will pass in a default way to rank the results as a `key` function into the sorted function (which is built into Python). The `key` function is a function that we can pass in to customize the way a list is sorted.

Now, let's build the restaurant class:

```
class Restaurant:

    all = []

    def __init__(self,name,stars):
        self.name, self.stars = name, stars
        Restaurant.all.append(self)

    def __repr__(self):
        return self.name
```

We don't yet need to actually make the `similar` yet, but let's just test if our program's basic function works.

```
>>> x = Restaurant('Thai Basil',5)
>>> y = Restaurant('Thai Delight',5)
>>> x
Thai Basil
```

And finally, we also need to figure out what it means to be similar to another restaurant.

```
    def similar(self,k,similarity=reviewer_overlap):
        "Return the K most similar restaurant to SELF."
        others = list(Restaurant.all) #We make a copy to not delete this restaurant from
        others.remove(self)
        f = lambda r: similarity(r,self) #f is a curried similarity function because sim
        return sorted(others, key=f, reverse=True)[:k]
```

Here's an idea for what `similarity` does: the same people reviewing 2 restaurants means their style may be pretty similar.

```
    def reviewer_overlap(r,s):
        "Number of users who reviewed both"
        return len([u for u in r.reveiewers if u in s.reviewers])
```

And we need to update our `__init__` function to include reviewers:

```
    def __init__(self,name,stars):
        self.name, self.stars = name, stars
```

```
        self.reviewers = reviewers #list of user_ids
        Restaurant.all.append(self)
```

Now we can't just keep using fake data. Let's load that real data from the JSON file:

```
import json

for line in open('reviews.json'):
    r = json.loads(line)
    business_id = r['business_id']
    if business_id not in reviewers_by_restaurant:
        reviewers_by_restaurant[business_id] = []
    reviewers_by_restaurant[business_id].append(r['user_id'])

for line in open('restaurants.json'):
    r = json.loads(line)
    reviewers = reviewers_by_restaurant[r['business_id']]
    Restaurant (r['name'],r['stars'],reviewers)
```

This program now works! But it's pretty slow. The big part of this is the `reviewer_overlap` function, since it goes through two lists twice. Thus, it is quadratic time.

Linear-Time Intersection of Sorted Lists

Let's optimize our code:

```
for line in open('restaurants.json'):
    r = json.loads(line)
    reviewers = reviewers_by_restaurant[r['business_id']]
    Restaurant (r['name'],r['stars'],sorted(reviewers))
```

Why does the `sorted` fix our code?

Well, let's take two sorted lists:

```
3,4,6,7,9,10
1,3,5,7,8
```

Well, since 1 is smaller than everything else in the first list, it can immediately skip to the second value. 3 matches, and it's smaller than 4, so the function now moves to 5. So on and so forth. You never need to check the entire list twice! It is linear because the maximum time is just the combined length of the two lists. We're always moving forward in one list or the other

Let's make a function that takes advantage of this:

```
def fast_overlap(s,t):
    """Return the overlap between sorted S and sorted T."""

    i, j, count = 0, 0, 0
    while i < len(s) and j < len(t):
    if s[i] == t[j]:
```

```
        count, i, j = count + 1, i + 1, j + 1
    elif s[i] < t[j]:
        i = i + 1
    else:
        j = j + 1
    return count
```