CSE231 - Lab 13

Inheritance, Method Overrides

What is inheritance?

Inheritance tends to be a confusing topic to many, but if you've been following along with the concept of classes so far, it should come as pretty intuitive.

Classes have one fundamental idea: sub (child) classes ***inherit*** *their super (parent) class' attributes and functions.*

Hopefully you can see why this is useful. In the same way we package bundles of code to reuse (functions), we can package an entire *class* to reuse.

Let's say we have a class named "`Car`", and two other classes, one named "`Ford`" and one named "`Tesla`".

The `Car` class might have attributes or functions that we want to pass down to the `Ford` and `Tesla` class, because both could be considered instances of type `Car`.

Inheriting From a Parent

```
class Car(object):

    ...

class Ford(Car):

    ...

class Tesla(Car):

    ...
```

Reusing a Constructor

Assuming our child class doesn’t need a unique constructor, (we’ll get to that in a bit), we can call the parent class’ constructor to initialize for us.

```
class Car(object):

    def __init__(self, owner, model, plate):

        self.owner, self.model, self.plate = owner, model, plate

class Ford(Car):

    def __init__(self, owner, model, plate):

        Car.__init__(self, owner, model, plate)
```

What is this?

```
Car.__init__(self, owner, model, plate)
```

It’s some funky syntax, right? We’re effectively saying:

1. Use `Car`’s constructor ...	`Car.__init__`
2. With `Ford`’s class ...	`(self,`
3. And pass `Ford`’s constructor arguments ...	`owner, model, plate)`

This is referred to as an *unbound call*. There is an equivalent way of doing this that you might prefer, albeit a bit more ambiguous than the one above:

```
super().__init__(owner, model, plate)
```

<sidenote>

I think it’s important to highlight the difference between this:

```
ClassName()
```

… and this:

```
ClassName
```

What I’m about to describe applies to function names, too. When you add those parentheses, you are *calling* that class -- i.e., you’re *instantiating* it. When you use the name *alone*, you are accessing the class’ reference. You’re taking the class and accessing its member *without* instantiating it. There’s much more on this topic that we could get into, but this is as far as we’ll go in this course.

Example Time

So all of this was just about *how* you sub-class. But now, let me show you why creating hierarchical structures like this can be useful.

We’re gonna be talkin about cars again, if you couldn’t already tell (again, please don’t ask me about cars -- I know nothing of the subject, they’re just really good to use as examples here).

L13-1

Method Overriding

When creating sub-classes, there are times where you might want to have the same function as the super-class, but slightly altered to cooperate with the sub-class.

We can achieve this via *method overriding*, essentially re-defining a method function, with the same name, but with a different algorithm. You can do this for *all methods*, user-defined or magic.

L13-2

Aaaaand that’s it!

Congratulations!

You’ve made it through CSE231...

It has been a rough, and strange semester. Many of you probably didn’t do as well as you hoped, and many of you probably would’ve preferred the in-class setting.

When you combine the difficulty of programming with the feeling of isolation (whether it be in the home you grew up in, or in your dorm on campus), I think the challenge ramps up significantly. Your mental health begins to deteriorate, and keeping up with course content becomes an excruciating challenge.

I’ve been talking with the professors about doing *something* to accommodate this strange time that we’re in, but I can’t make a guarantee that anything will happen, as unfortunate as that may be.

Why is programming so difficult?

Programming, as you probably came to discover, is immensely difficult. It’s not something that we, as humans, are “naturally” good at. It has many similarities to mathematics, both in concept, and intuitiveness.

Hopefully, you came to discover that the actual “programming” part of it was easy, with the real challenge being the problem-solving aspect.

And, this is what real-world computer science is -- designing programs to solve problems, or make products. NASA uses Python for a ton of their projects, and Netflix uses Python to run their databases, just as an example.

Let’s be real here

And look, if you’re a computer science major, and didn’t do well in this class, I think you’re going to have a hard time in the future unless you change something right now.

Dr. Nahum, (the CSE232 professor), will give you a million different coding problems to solve, and with a *much* harder language (C++). You’ll be learning more about references, you’ll have pointers to deal with, memory allocation, RAM, etc. Dr. Nahum is also famous for handing out *tons* of Academic Dishonesty Reports. If CSE231 is the “weeder” class, then CSE232 is the excavator.

I think that, if you felt satisfaction in completing CSE231 projects, or found the challenge of them “fun”, you’re going to have a great time as a computer scientist. But if you *didn’t* feel either of those, I think you probably came into this class thinking that computer science was something else.

For you engineers out there

And as for the engineers, you will *probably* see another programming language in your lifetime. Many engineering majors have to take a course in MATLAB or R (two other programming languages), it’s just that you won’t be bogged down by so many computer science-y topics (anything having to do with classes, mainly).

Why is this? Because programming is *immensely* powerful for doing calculations, modeling, graphing, and so much more (a lot of what you do as any engineer). Programming is like mathematics but without pencil and paper.

(And, if you’re comfortable enough, you can just start making Python scripts to make your physics homework easier)

What we didn’t cover

There is *a lot* we didn’t cover in this course. You will learn the rest in CSE232, and your 300-levels...

- Pointers (there are no pointers in Python, lucky for you guys)
- Lambda Functions (you might’ve seen these before, briefly)
- Regex
- Advanced Data Structures
- Time Complexity
- Recursion, the Call Stack
- Famous algorithms (sorting algorithms, Dijkstra's algorithm, A* search, tree traversals, hashing)
- Git, Github (source control)

What we didn’t cover… in Python

There are a lot of Python-exclusive features we haven’t talked about, too. I would encourage you to look into these on your own time!

- `map()`, `zip()`, `filter()`, `exec()`, `eval()`, `all()`, `any()`, ...
- `yield`, `from`, `with`, `as`, `nonlocal`
- Nested Functions
- Generator Functions
- Variable arguments, variable keyword arguments
- Complex Numbers
- Decorators
- Walrus Operator (`:=`)
- Bitwise Operators (>>, <<, &, |, ^, ~) (useful in CSE320!)