Good Software Practices

Making coding that much more fun!

What we will discuss?

- Object Oriented Programming
- Project Structure
- Design Patterns
- Unit Testing (if time permits)

Motivation

- How often do you re-use code?
- How much of your code is actually duplicated?
- Does your project directories typically look like a huge mess?
- How long does it take you to change functionality?
- When you change functionality, does your code break down the line?

Then you’re in the right talk!

Scenario

- You are making a library of optimisation algorithms
- There are lots of different types of algorithms but you start off with gradient descent
- You are building a classifier using logistic regression
- It looks something like this (see example code)

Why is this so bad?

- No documentation: It's hardly clear to me what some things mean
- Not generalisable: See optimise_logistic_loss()
- A complete mess: All code is in one file, there is no modularity

OOP to the rescue!

Why OOP?

- Duplicate code is bad? Who wants to do things more than once?
- Change is inevitable? And the headache that comes with it…
- Abstraction is a computer scientist’s friend

So what is new in OOP?

- In OOP *everything* is an Object —> everything is different?

- Well not really, but some things are and taking advantage of them is great

Abstraction?

- For the maths folk: When I have a function f: R -> R that is differentiable, I don’t need to know the exact form of f to know that it is guaranteed to have a derivative

- That’s an abstraction: ignoring the details and focusing on what matters for the problem at hand

- For the non-math folk: Dogs and cats -> animals, therefore they both make sounds (“woof”, “meow”)

Object

Object

Methods (Functions)

Object

Methods (Functions)

Variables (State)

Object

Methods (Functions)

Variables (State)

Inheritance (Behaviour)

So what does this mean?

- An object has its own functions e.g. cat.make_sound()

- An object has its own states e.g. cat.fur_colour or cat.age

- An object has a family tree that defines its behaviour e.g. a cat is a mammal, a mammal is an animal, an animal is a living_being —> a cat can breathe!

Back to Optimisation

Q: “Isn’t it weird to think of an optimisation *algorithm* as an object?”

Back to Optimisation

Q: “Isn’t it weird to think of an optimisation *algorithm* as an object?”

A: “Yes. But it is. Get over it and join the club.”

Back to Optimisation

Q: “Isn’t it weird to think of an optimisation *algorithm* as an object?”

A: “Yes. But it is. Get over it and join the club.”

Q: “So what does an optimisation algorithm have?”

Back to Optimisation

Q: “Isn’t it weird to think of an optimisation *algorithm* as an object?”

A: “Yes. But it is. Get over it and join the club.”

Q: “So what does an optimisation algorithm have?”

A: “It depends on the application.
OOP needs planning. ”

What would you give an Optimisation_Algorithm?

(Audience participation encouraged)

What did we learn?

- Optimisation_Algorithm is an *abstract* class
- It defines an *interface* for *concrete* classes
- Any concrete class knows that it **must** implement do_iteration()
- But all the other code is the same for any other implementation
- Gradient_Descent and SGD both only need to implement do_iteration(), the rest is *inherited* —> Code reusability!

Cost Functions

Q: “Surely cost functions should be functions, right?
The name is a big hint… ”

Cost Functions

Q: “Surely cost functions should be functions, right? The name is a big hint… ”

A: “NO! Wrong! Have you learned nothing?! ”

Cost Functions

Q: “Surely cost functions should be functions, right?
The name is a big hint… ”

A: “NO! Wrong! Have you learned nothing?! ”

A: “Imagine a Gaussian distribution. It could have
as its state the mean and covariance matrix
and then could take as an argument a vector x.
It could give the density or the cumulative probability etc.”

What does main look like now?

So what have we achieved?

- Now we can write up a new optimisation algorithm much more quickly. Maybe 20 lines of code instead of 100.

- Can now write scripts with any combination of these cost functions and algorithms.

- To change any functionality, we can do so in as few classes as possible and with minimal impact to the rest of the code. The code is now *decoupled*.

Design Patterns

- *“Code should be open to extension but closed for modification”* - Head First Design Patterns

- *“Program to an interface, not an implementation”* - Head First Design Patterns

Template Method

- *"The Template Method Pattern defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure."* - Head First Design Patterns

- This has already been seen with the do_iteration() in the Optimisation_Algorithm class.

- This is done at compile-time by subclassing.

- The template class provides the template algorithm and the concrete classes provide the differences.

Factory Pattern

- A class that creates different types of other concrete classes
- In this case we have a Data_Creator_Factory
- If any new Data_Creators are developed, the only code that has to change is that in the Data_Creator_Factory

Strategy Pattern

- Defines a set of algorithms, which are encapsulated (into classes) and are interchangeable i.e. they conform to the same interface. Instead of inheritance, uses delegation.

- Example, suppose in SGD we would like to actually sample with something other than a uniform distribution.

- Optimisation_Algorithm could instead have as a member of the class a Random_Sampler.

- Upon creation we could pass in the Non-uniform_Random_Sampler to SGD.

- Algorithm chosen at run-time using containment. In this case the algorithm is the Random_Sampler and this can be chosen at run-time. The Optimisation_Algorithm delegates the sampling to the Random_Sampler.

But what happens to Gradient_Descent?!

There’s nothing random about it!

Null Object Pattern

- Fear not! The Null Object can be used.
- An object that basically is a placeholder and nothing more.
- So in the place of Random_Sampler for Gradient_Descent, we could create a class that returns all indices.
- Has the same interface as Random_Sampler *but it acts as if it were not there*.

Command Pattern

- Suppose we would like to run a bunch of experiments and we have lots of machines to do so but the requests are done asynchronously.

- We could have a queuing system for each experiment and different machines could pop the latest experiment off the queue and then perform the experiment and save the results somewhere.

- For example, we want to run lots of experiments on different data with different cost functions.

- Just create an object that encapsulates this information.

- When the command is put into the queue and it is its turn to be executed —> command.execute().

There are plenty more design patterns!

But such a small amount of time :(

Anything else?

- Make a package that has common functions that you literally *always* use and call it auxiliary or something. Then you don’t have to rewrite the sigmoid function 9208308 times in your sad PhD life.

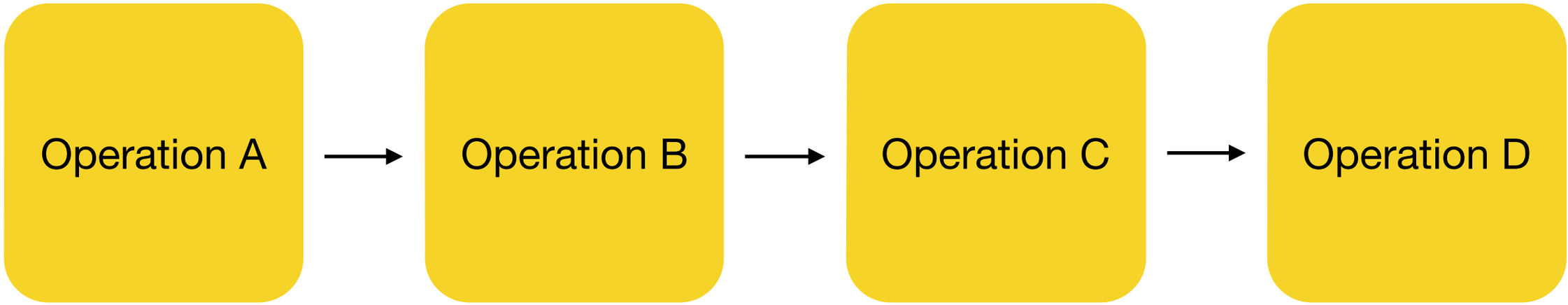
- Documentation! Just do it, it means that that input to that arbitrarily named method has some meaning a week down the line.

- Oh and…

Unit Testing!

(as fun as it sounds)

Why unit test?



How many possible ways are there to go wrong?
How does this grow with the number of operations?

Unit Tests

- Test everything! Every possible way an algorithm could fail.
- E.g. What happens to your sorting algorithm when it is given an empty list, or a list of strings, or when every item is equal, etc.
- Only need to write the tests once.
- Every time you *change* something (remember, that is the only constant in software development), just run the tests again and make sure they pass.
- Some people even write the tests first as it helps you think about the problems that could occur.
- Unit tests are the best type of documentation (said Matt who was quoting someone).

The end