(Software) Design Patterns

(as opposed to interior design patterns)

What we will discuss?

- Why we should use design patterns
- Various design patterns that I find useful

Motivation

- How often do you re-use code?
- How much of your code is actually duplicated?
- 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!

Before we start…

- I will use the words: Interface and Abstract Class interchangeably.

- I will not use their language specific definitions.

- My definition: An abstract type that is used to specify the particular methods that must be implemented by subclasses and variables that will be inherited.

- Any problems with that?

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 logistic regression classifier
- It looks something like this (see example code)

Why is this so bad?

- Not generalisable: See optimise_logistic_loss()

- A complete mess: All code is in one file, there is no modularity

- Change would lead to broken code.

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

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”)

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

- Ways to solve common problems that arise in OOP by following basic class blueprints.

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 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/composition.

- 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 Nonuniform_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 the information needed for one experiment.

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

Observer Pattern

- 2 Object types: Subject and Observer. The observers are notified about the subject. One-to-many relationship.

- The subject registers observers and whenever its state is changed, it informs them.

- Useful in UI design e.g. suppose we are creating a graph of cost per epoch and one for accuracy per epoch which dynamically change after every epoch.

- After each epoch, the Optimisation_Algorithm could let the graphs know that its state has changed and they could then check its state.

- That way if we want to make another graph later, just need to add another observer.

Singleton Pattern

- A class that can only be instantiated once in the whole program.
- Like a glorified global variable, *but only gets instantiated when needed.*
- Suppose we want statistics on the whole program e.g. a logger or an object that has the configurations needed for the program.
- We only want one of these that all objects use, *so we need a global access point.*
- This pattern ensures that this is the case. Unfortunately, this isn't that easy in Python because there is no such thing as a private constructor, but it's worth knowing for other languages.

There are plenty more design patterns!

But such a small amount of time :(