3 Great Design Patterns for Data Scientists

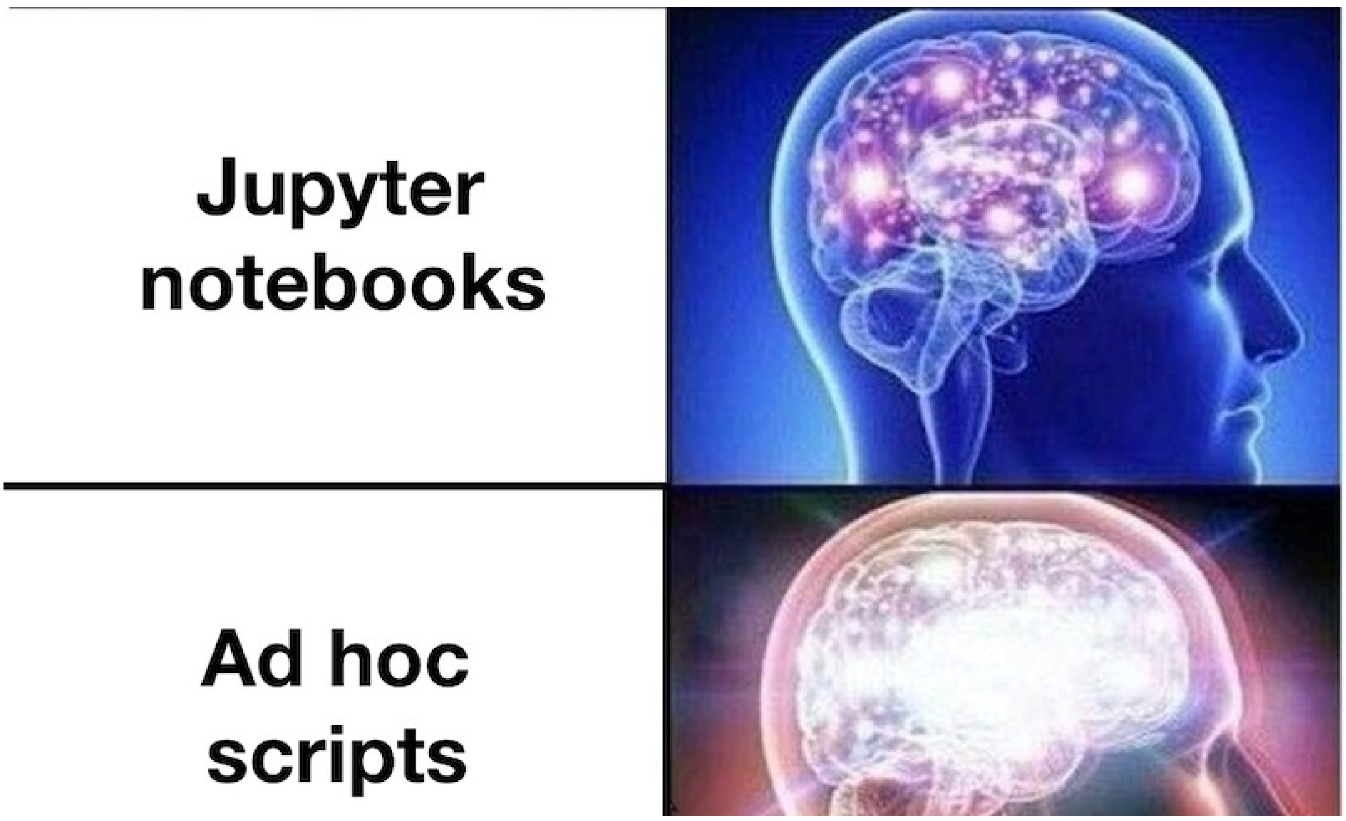
Want to learn how to write better data science code? Use design patterns to write clean, maintainable, testable code.

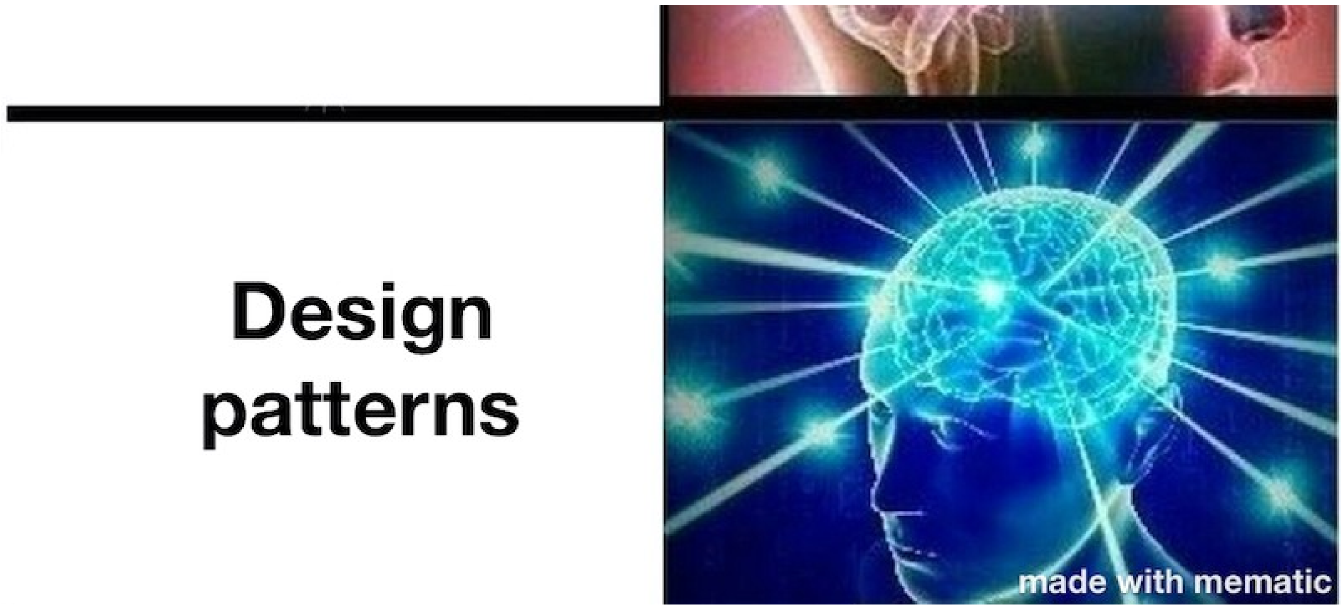
Richie Frost
May 30 · 5 min read ★

Introduction

When writing code as a data scientist, your goal is often to write things quickly so that you can vet whether or not something is a good idea before you get too far down the road. Nobody likes to spend months working on a project only to find out that it's garbage.

So you write your code as quickly as possible when prototyping. But what happens when your just-get-it-working-for-now code isn't cutting it anymore, and your code needs to be more robust and maintainable? This is where design patterns come in handy.





Meme by Richie Frost, created using Mematic

What are design patterns?

To put it simply, design patterns are common solutions to common problems when writing software. What makes them so great is that they're so universally applicable, but you have to know how to apply them. You can learn more in-depth about some common design patterns here.

Why use design patterns?

I can think of a couple of reasons that I love using them.

- Reduce cognitive load while coding
- Spend less time debugging
- Code is much more testable
- Easier to build reusable tools

So, without further ado, let's get into 3 great design patterns for data science workflows.

1. The Builder Pattern

What is it?

The builder pattern is a flexible way of creating complex objects, especially when these objects share a lot of similarities but have a lot of optional parameters. The builder pattern takes the object construction logic out of the object itself, and instead creates relevant properties for the object on the fly — often by using the method chaining technique. The key to enabling method chaining is to return the object itself from methods used to build the object you want, so that chained methods can modify the same object.

Example: Generating SQL queries

I write a ton of SQL queries day to day, and found that there's a lot of similarity in structure to most of my queries. However, writing them by hand is a fairly error-prone process and creates a lot of duplicated code. So rather than writing dozens of individual queries, I use the builder pattern to generate queries for me. This also comes in handy a lot when I write big, nasty queries with nested select statements and multiple joins, where it's easy to get lost in the weeds and make mistakes when writing queries by hand. Not to mention this method is easily testable, whereas writing SQL queries by hand is harder to test!

Let's write a simple query builder to illustrate how this pattern can be useful.

I first initialize the builder with the base table from which I'll be selecting tuples. Then I can add columns to select, 'group by' clauses, joins, and 'where' clauses as I need them. This is overkill for a simple "SELECT * FROM foo" type of query, but these building blocks make it easier to build more and more complex queries.

Here's an example of using the builder pattern to make a simple SQL query generator:

```python
class QueryBuilder:
    def __init__(self):
        self.select_value = ''
        self.from_table_name = ''
        self.where_value = ''
        self.groupby_value = ''

    def select(self, select_arg):
        self.select_value = select_arg
        return self

    def from_table(self, from_arg):
        self.from_table_name = from_arg
        return self

    def where(self, where_arg):
        self.where_value = where_arg
        return self

    def groupby(self, groupby_args):
        self.groupby_value = groupby_args
        return self

    def build(self):
        if self.where_value:
            where_clause = f'WHERE {self.where_value}'
        if self.groupby_value:
            groupby_clause = f'GROUP BY {self.groupby_value}'

        return f"""
            SELECT {self.select_value}
            FROM {self.from_table_name}
            {where_clause}
            {groupby_clause}
        """

# Simple builder pattern example for building queries
query = QueryBuilder()
query.select('Customer, Region, SUM(SaleValue)') \
    .from_table('Sales') \

    # Optional - add where and groupby clauses.
    # Object construction can easily be either simple or complex
query.where('DATEDIFF(day, TimeStamp, CURRENT_TIMESTAMP) < 180') \
    .groupby('Customer, Region')

# Builder pattern collects the optional arguments and builds the actual SQL text
query_text = query.build()

"""
query_text value:
SELECT Customer, Region, SUM(SaleValue)
FROM Sales
WHERE DATEDIFF(day, TimeStamp, CURRENT_TIMESTAMP) < 180
GROUP BY Customer, Region
"""
```

Using the builder pattern to generate SQL queries

2. Dependency injection

What is it?

In its simplest form, dependency injection is when you insert the thing you're depending on as an argument. Don't know which database class to use? Your function doesn't need to know how the database class works, just that it does. Passing in the database class instance as an argument makes it easier to maintain — you can use any kind of database class that follows the same interface.

Without using dependency injection, you'll have a much harder time maintaining critical infrastructure like database classes.

One other great benefit of using dependency injection is that your code is much easier to write tests for. Just write a mock class (i.e. a mock database class) and use that in your tests, rather than having to use code that runs HTTP requests and slows down tests, for example.

Example: Database classes

My team uses both SQL Server and Cosmos DB, as well as other data sources. Passing in the database class as an argument makes it easy to swap out different databases for different ideas, and makes writing testable code a lot easier, since database classes are easy to mock.

Here's a simple example of using dependency injection:

```
# Don't do this
def get_data_bad(query_text):
    db = SQLDB()
    return db.get(query_text)

# What if you need to use a DocDB instance? Or a DynamoDB instance?
# Do this instead
def get_data(db, query_text):
    return db.get(query_text)

# Example
sqldb = SQLDB()
query = 'SELECT * FROM Foo'
data = get_data(sqldb, query)

# Or, if you need to use DocDB instead, you don't need to change your original get_data method
docdb = DocDB()
query = 'SELECT c.* FROM c'
data = get_data(docdb, query)
```

Using the dependency injection pattern to get data from a database

3. The Decorator Pattern

What is it?

The decorator pattern is useful when you want to do something before and/or after a function, but don't want to modify the function itself. Essentially, what you're doing is capturing some state before your function runs, then capturing some state after it's done. This becomes very apparent when you have dozens of functions to modify in the same way, but can't afford to change them individually.

Example: Logging function metadata

Things that I've found useful are how long the function runs, the function's name, and sometimes different features about the output. Thankfully, Python functions are objects, so you can use the '@' decorator syntax for this pattern. All you need to do is create a function that wraps an inner function, then place the @my_decorator_name decorator before the function you want to decorate.

It's easier to see an example than to explain it with plain English :)

I won't get too deep into how decorators work in Python, but RealPython has a great article I highly recommend as a primer.

Reusing some of the code from the dependency injection example, we can time how long our database transaction would take:

```
from time import time

def log_time(func):
    """Logs the time it took for func to execute"""
    def wrapper(*args, **kwargs):
        start = time()
        val = func(*args, **kwargs)
        end = time()
        duration = end - start
        print(f'{func.__name__} took {duration} seconds to run')
        return val
    return wrapper

# Example usage
@log_time
def get_data(db, query):
    """Gets data from a SQL-based database"""
    data = db.get(query)
    return data

if __name__ == '__main__':
    # Decorated function will print 'get_data took X seconds to run'
    db = SQLDB()
    query = 'SELECT * FROM foo'
    data = get_data(db, query)
```

timer.py hosted with ♡ by GitHub view raw

Using the decorator pattern to time a function

Putting it together

Design patterns make for very reusable code, and you can put pieces together like building blocks to make your work a lot easier as a data scientist. For example, I'll often combine all three of these patterns to write queries to a database and see how long the query took in order to know if I need to optimize.

Conclusion

In this article, I've shown three ways to use design patterns as a data scientist for more robust, maintainable code. When you use design patterns in data science, your code quality goes up, your maintenance is easier, and your results are easier to reproduce and share.

Data Science Python Software Development Towards Data Science