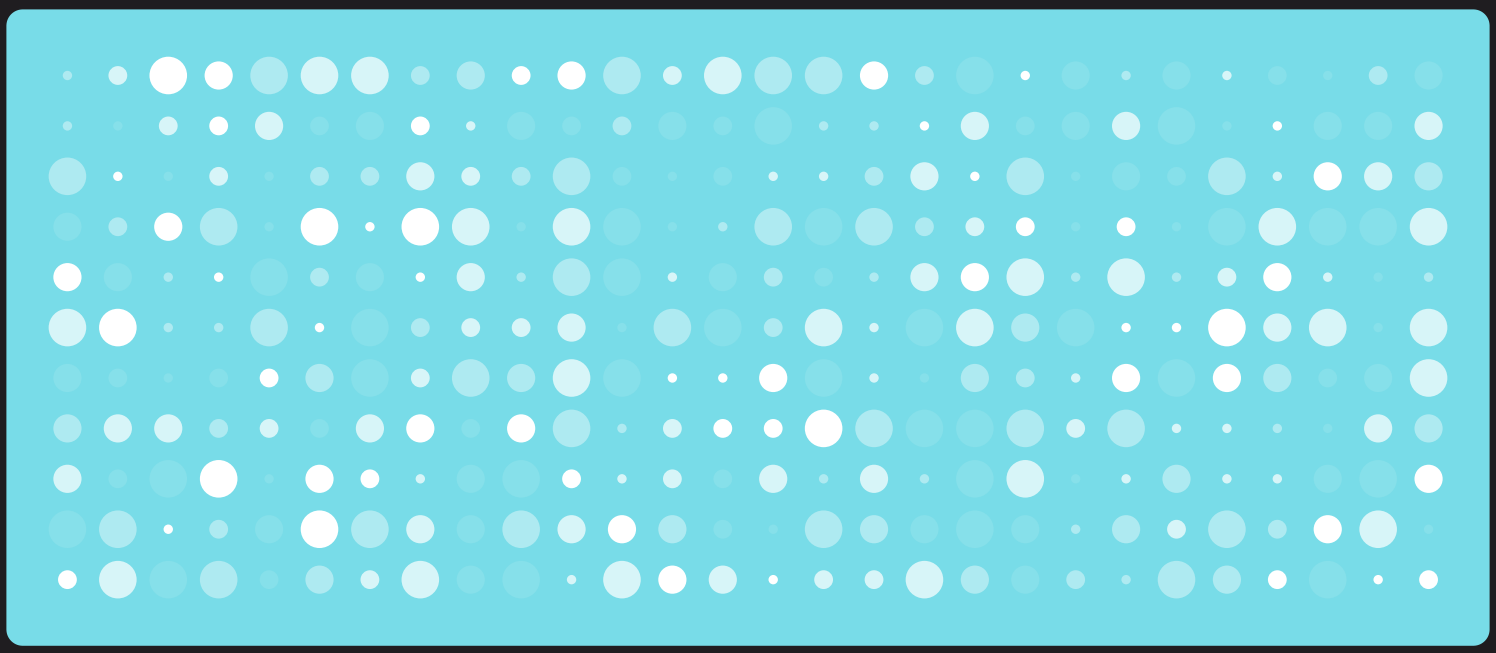

Elliott Phillips

Data Scientist
BSc OR III

Better Python

Software development & design introductory handbook

Last updated: January 4, 2022

Better Python: software design-focused learning materials to build maintainable, *extendable* code.

ABOUT

The materials enclosed provide a modern take on software design, using Python to develop a better designer mindset, consistently make better design decisions, refactor existing code, and create new software that is easy to change and scale.

2.1 Editor

2.1.1 Configuration

Settings used for Python development in VS Code:

```
{
    "editor.defaultFormatter": "esbenp.prettier-vscode",
    "[python]": {
        "editor.defaultFormatter": null
    },
    "editor.formatOnSaveMode": "file",
    "editor.formatOnSave": true,
    "editor.codeActionsOnSave": {
        "source.organizeImports": true
    },
    "python.linting.pylintEnabled": true,
    "python.linting.mypyEnabled": true,
    "python.linting.enabled": true,
    "python.analysis.typeCheckingNode": "strict",
    "python.formatting.provider": "black",
    "vim.smartRelativeLine": true
}
```

Placing this `settings.json` file in the project workspace means any other developer that directly works on this code will also have the same editor settings, which is convenient.

As an automatic code formatter, *Black* is very opinionated and works well for my use and preferences.

2.1.2 Linting

Using *Pylint*, we can create a customg linting configuration dotfile, `.pylintrc`.

```
[MASTER]
docstring-min-length=10
disable=missing-function-docstring,
        missing-class-docstring,
        missing-module-docstring
[FORMAT]
good-names=i,j,x,y,id
```

PATTERNS

Design patterns were conceived in the 90s when object-oriented programming was extremely popular. So naturally, design patterns rely on classes and inheritance quite a lot. However, programming languages have evolved: Python not only has classes, but also tuples, dictionaries, Protocols and dataclasses.

3.1 Analysing the Factory pattern

The factory allows you to inject objects of a certain subtype into a part of an application that then uses those objects without knowing what they are exactly.

Doing this helps reduce coupling by enabling you to introduce and inject new kinds of objects into that same application without having to change the code. A related principle is the *"single responsibility principle"* enforcing the idea of not both creating and using something in the same place – those are two separate responsibilities.

The Open Closed principle also plays an important role in the factory pattern: we want to be able to extend the application without having to extensively change the code. The code should be *open for extension but closed for modification*. The factory pattern achieves this by letting you, the developer, introduce new exporter factories without modifying the original code interface.

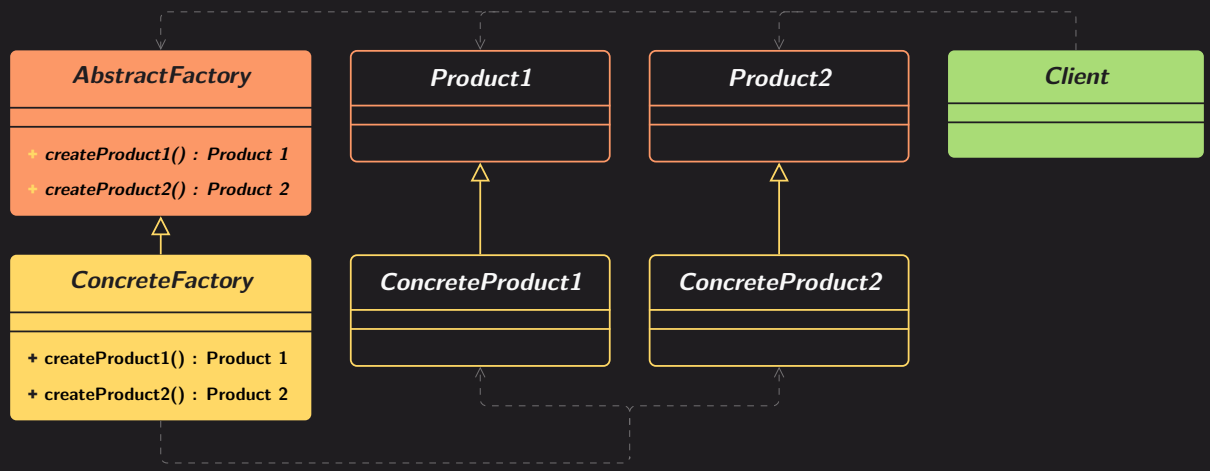

Figure 3.1: *Factory pattern Unified Modeling Language (UML) diagram, demonstrating separation of creation from use.*

3.2 Factories

In this section, we take a look at the factory design pattern and strip it down completely until we arrive at the design *principles* that are behind the pattern. The most important design principle behind the factory pattern is to separate creation from use.

3.2.1 A more Pythonic Factory pattern

In this example, we have `VideoExporter` and `AudioExporter` Abstract Base Classes (ABC) that cover various output formats, and an `ExporterFactory` – which is also an ABC – that has abstract methods for creating the video and audio exporters.

```python
class VideoExporter(Protocol):
    """Basic representation of video exporting codec."""

    def prepare_export(self, video_data: str) -> None:
        """Prepares video data for exporting."""
        raise NotImplementedError

    def do_export(self, folder: Path) -> None:
        """Exports the video data to a folder."""
        raise NotImplementedError
```

This user can choose to use the fast, high or master quality exporter to render their video and audio files, of which these are subclasses of the exporter factory. To facilitate these different factories, we create an object a dictionary `FACTORIES` and define a method `read_factory()` to read the user's desired output quality as input, get the corresponding factory to use the appropriate video and audio exporters, and then prepares and does the export.

```python
FACTORIES = {
    "low": FastExporter(),
    "high": HighQualityExporter(),
    "master": MasterQualityExporter(),
}

def read_factory() -> ExporterFactory:
    """Constructs an exporter factory based on the user's preference."""

    while True:
        export_quality = input(
            f"Enter desired output quality ({', '.join(FACTORIES)}): "
        )
        try:
            return FACTORIES[export_quality]
        except KeyError:
            print(f"Unknown output quality option: {export_quality}.")
```

Demonstrative materials.

4.1 Code blocks

Syntactic sugar for relative file paths:

```
function fn(x: string | number) {
  if (typeof x === "string") {
    x = SomeClassObject() // do something
  } else if (x >= 3.14159265) {
    "42" // do something else
  } else {
    `The answer is ${x * 1.414}`; // template literal!
  }
}
```

Inspecting particular lines within a code file:

```
class ExporterFactory(ABC):
  """
  Factory that represents a combination of video and audio codecs.
  The factory doesn't maintain any of the instances it creates.
  """

  @abstractmethod
  def get_video_exporter(self) -> VideoExporter:
    """Returns a new video exporter belonging to this factory."""

  @abstractmethod
  def get_audio_exporter(self) -> AudioExporter:
    """Returns a new audio exporter belonging to this factory."""
```