Day 2: Abstraction and composition

Hamish Gibbs

Writing re-usable code

- Code should be like a recipe.
  - Generally: good code tells *how* to do something, not *what* you've done.
- Scripting vs. programming
  - **Scripting:** Small bits of code that do a single thing.
  - **Programming:** General-purpose "recipes" for transforming inputs to outputs.

Example: scripting in python

- A simple script for converting Fahrenheit to Celsius

```
1  temp_f = 100
2
3  temp_c = 5/9 * (temp_f - 32)
```

- What's wrong with this?
  - Nothing, it works.
  - But what if we want to change the value of `temp_f`?
  - What if we want to convert multiple Fahrenheit values to Celsius?

Solution: Abstraction

- We want to **abstract** the logic that converts temperatures into a “recipe” with:
  - **Input:** any value in Fahrenheit.
  - **Output:** the converted value in Celsius.
- Our “recipe” can be written as a **Function**.

Example: programming in Python

- A **function** for converting temperatures:

```
1 def convert_f_to_c(temp_f):
2     return 5/9 * (temp_f - 32)
```

- Now, our logic can be applied to multiple values:

```
1 print(convert_f_to_c(100))
2 print(convert_f_to_c(120))
```

- Or we can apply our to function to a `list` of values:

```
1 temps_f = [100, 120, 80]
2 temps_c = [convert_f_to_c(x) for x in temps_f]
```

Functions

- **Functions** are a named bundle of logic.
  - I think of a function as a “pipe” that transforms values into other values.
- Example functions (*Tip: useful for the challenge!*):
  - `model = fit_model(train)`
  - `fig = plot_scatterplot(data)`
  - `save_image(img, path)`
- Another analogy: think of functions as the “verbs” and variables as the “nouns” of your program.

Composition

- Functions help to break up your code into small, reusable “modules.”
- These modules can be **composed** together:

```
def convert_multiple_f_to_c(temps_f):
    return [convert_f_to_c(x) for x in temps_f]
```

- Programming is less about tricky logic problems, more about abstraction and composition.

Scripting vs. Programming

- The line between **scripting** and **programming** is fuzzy.
- Often, you need to re-use bits of a script, so you start re-writing it into functions.
- If these functions are useful enough, you can incorporate them into a library.
  - My own example of this (in R): ggutils.

Classes: logic + data

- **Functions:** logic (a “recipe”)
- **Variables:** data (actual “values”)
- **Classes:** An **abstraction** for combining data and logic.

Classes

- Classes have two components:
  - **Attributes:** data.
  - **Methods:** functions.

```
    class WeatherStation:
        def __init__(self, temps_f): # Default initialization method
            self.temps_f = temps_f # an "attribute"

        def convert_f_to_c(temp_f): # A "method"
            return 5/9 * (temp_f - 32)

        def convert_temps_f_to_c(self): # Another "method"
            return [self.convert_f_to_c(x) for x in self.temps_f]
```

- Now, my functions are directly **coupled** to my data and I have given this `Object` a name: `WeatherStation`.

Using a class

- A **class** is a general purpose construct, like a **function**.
- We have to initialize our class with some data:

```
1 station = WeatherStation(temps_f = [100, 120, 80])
```

- Here, `station` is an **instance** of the class `WeatherStation`.
- Then we can use the methods of the class for this instance:

```
1 print(station.convert_temps_f_to_c())
```

Inheritance

- Classes can be **extended** to represent different objects objects with the same **interface**.
- Here, the `WeatherStation` has a general purpose method `get_temperatures` which should always return the temperature in Celsius.

```
    class WeatherStation:
        def __init__(self, temps):
            self.temps = temps

        def convert_f_to_c(self, temp_f):
            return 5 / 9 * (temp_f - 32)

        def get_temperatures_c(self):
            return self.temps
```

Inheritance

- We could create two child classes which **inherit** the `WeatherStation` interface.
- Assuming an `AmericanWeatherStation` is always initialized with `temps` in Fahrenheit:

```
    class AmericanWeatherStation(WeatherStation):

        def get_temperatures_c(self):
            return [self.convert_f_to_c(x) for x in self.temps]
```

- Assuming a `EuropeanWeatherStation` is always initialized with `temps` in Celsius:

```
    class EuropeanWeatherStation(WeatherStation):

        def get_temperatures_c(self):
            return self.temps
```

Inheritance

- Inheritance gives a common **interface**.
- Now, I can write a function that consumes any `WeatherStation` object.

```
    def get_total_temp_c(station):
        return sum(station.get_temperatures_c())
```

Tutorial #1: Functions

- Functions
- Core concepts:
  - Using built-in functions (and the standard library)

```
1 import math
2 math.log10(10)
```

  - Writing your own functions

```
1 def add_3(x):
2     return x + 3
```

  - Composing functions

```
1 def add_5(x):
2     return add_3(x) + 2
```

Tutorial #2: Functions (Optional)

- More control flow tools §4.7-4.8 (Optional)
  - *This is more of a deep dive. If you feel shaky with the basics of functions, jump ahead to classes, then return to this!*
- Core concepts:
  - Default arguments

```
1 def add(x, y = 2):
2     return x + y
```

  - Keyword arguments

```
1 add(4, x=4) # Error: duplicate value for the same argument
```

Tutorial #3

- Object-oriented programming
- Core concepts:
  - Writing custom classes

```
1 class PartyAnimal:
```

  - Initializing classes

```
1 an = PartyAnimal()
```

  - Class inheritance

```
1 class CricketFan(PartyAnimal):
```

Tutorial #3: possible pitfall

- Tutorial #3 includes the following code:

```
1 from party import PartyAnimal
```

- This requires actually breaking our code into different scripts (`.py` files).
- We can't do this because we are still using Colab.
  - For now, just carry on in the same Notebook.
  - We will introduce `.py` files this afternoon!