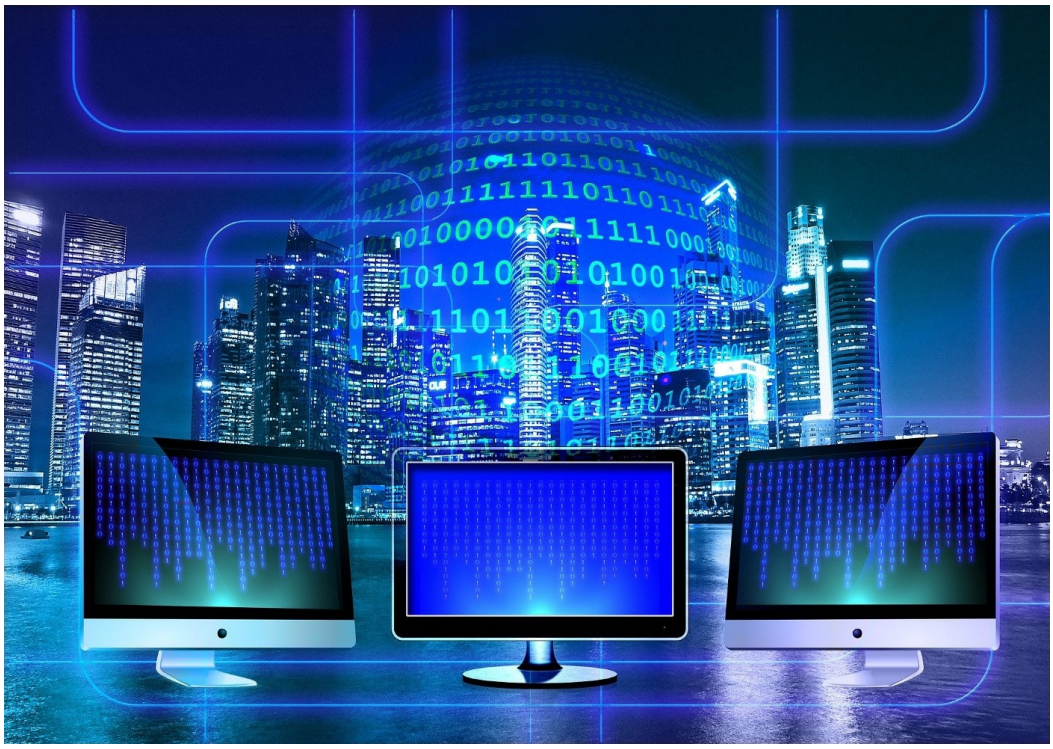
towards
data science

Polymorphism in Python: Probability Analysis

Object-oriented programming using Python

Michael Grogan 2 days ago · 5 min read★



Source: Photo by geralt from Pixabay

Data scientists often face the accusation of writing what is known as “spaghetti code”. That is to say, code which technically accomplishes a task but is not always reproducible or laid out in the best manner.

From this point of view, **polymorphism** — a core concept in object-oriented programming — has great value in allowing for implementation of multiple functions within a particular class. This makes it a lot easier for users to be able to use similar functions alongside each other, while also ensuring that the code is maintainable by grouping such functions together.

While this can sound somewhat abstract to someone without a computer science background — one of the main functions of object-oriented programming is to ensure that the code is organised and reproducible.

This saves a lot of time in ensuring maintenance as a program invariably increases in size.

Let's see how this works by using polymorphism to implement a range of probability calculations in conjunction with numpy.

Probability analysis

In this example, let's take a look at how polymorphism can be used to store numerous probability calculations within a larger class.

Firstly, let's define a broad class which we will use to define various "sub-classes" of probability calculations:

```
class Probability:
    def __init__(self, name):
        self.name = name

def __str__(self):
        return self.name
```

Binomial Probabilities

A binomial probability is a discrete probability that calculates the probability of a certain number of successes given a certain number of trials.

numpy.org gives an example of such a calculation, whereby one is asked to calculate the probability that of nine oil wells drilled with a success probability of 0.1, all of these wells will fail.

Now, imagine this scenario. Suppose that we wish to create a program that stores calculations across various probabilities in separate functions — each contained within the one class. In addition, calling the class allows us to define the number of trials that we wish to calculate when attempting to find the cumulative probability of failure.

The class is defined as below, with separate functions containing probabilities of **5%**, **10%**, and **15%** nested within the class:

Note that *self.length* indicates the number of trials that we will specify when calling the function.

For example, let's specify 10 trials and calculate the probability of failure across the desired probabilities:

```
>>> a = Binomial(10)
>>> print(a)
>>> print(a.fact())
>>> print(a.binomial5())
>>> print(a.binomial10())
>>> print(a.binomial15())

Binomial
Probability of failure according to the Binomial Distribution.
0.5
0.6
0.1
```

We see that at a 5% chance of success, the probability of failure is 50%. At 10%, it is 60% while at 15% it goes down to 10%.

How about increasing the number of trials to 1000?

```
>>> a = Binomial(1000)
>>> print(a)
>>> print(a.fact())
>>> print(a.binomial5())
>>> print(a.binomial10())
>>> print(a.binomial15())

Binomial
Probability of failure according to the Binomial Distribution.
0.623
0.363
0.259
```

At 5%, the probability is now 62.3%. At 10, it is 36.3% while at 15% it is 25.9%.

From a real-world standpoint, one might wish to devise a computer program that can spontaneously generate numerous failure probabilities across a large number of trials, and parameters other than the probability rate may also vary. Using polymorphism can allow for flexibility in being able to do this.

Pareto Distribution

Now, consider this scenario. Suppose that we wish to generate samples from a Pareto Distribution. A Pareto distribution is a power law distribution whereby the tails of the distribution are heavy. In other words, much of the data is contained in the tails of the distribution:

Source: Jupyter Notebook Output

Specifically, let’s generate three separate distributions with shape 3 and modes 1, 2, and 3.

As we can see, the shape of the Pareto distribution is constant while the modes are varied. Each mode (mode1, mode2, mode3) is defined as a function whereby the value of the mode is different. Again, *self.length* defines the number of trials that we wish to generate.

```
>>> b = Pareto(10)
>>> print(b)
>>> print(b.fact())
>>> print(b.mode1())
>>> print(b.mode2())
>>> print(b.mode3())

Pareto
Sample generated in accordance with the Pareto Distribution.
[1.36412427 1.10291466 1.47183876 1.66282503 1.18855344 1.08670931
 1.01762507 1.04154705 1.48818185 1.47923808]
[2.28969839 3.64922758 3.48322786 3.71563878 2.02763407 4.14306486
 2.04481284 3.57690852 4.38221439 3.50612773]
[3.89360177 5.44499969 3.00307348 8.11591885 3.83083417 4.82244346
 5.91314304 3.16790445 3.06548931 3.91090296]
```

Let’s try 1,000 trials.

```
>>> b = Pareto(1000)
>>> print(b)
>>> print(b.fact())
>>> print(b.mode1())
>>> print(b.mode2())
>>> print(b.mode3())

Pareto
Sample generated in accordance with the Pareto Distribution.
[ 1.04522737  1.13748388  2.64986056  1.01188388  1.92169205  ...
  1.56696587  1.1202479   2.08817365  1.46907723]
[ 2.59499208  2.2019354   3.78301714  3.33741161  2.08731817  ...
  3.23218408  2.48797598  2.47920052  2.2045201 ]
[ 4.44148781  7.86723551  6.78859869  3.24117621  3.47112606  ...
  3.03165741  3.20184443  4.11208707  3.37364986]
```

As we can see, polymorphism provides more of a structure to an array of functions by allowing different classes to be stored another a larger class with a single name.

While in this example, a handful of functions could technically be run separately, this becomes a significant problem in the context of a computer program with thousands — maybe even millions of functions.

Using polymorphism to provide a structure to such functions allows for the code to be better maintained and increases efficiency when it comes to putting programs into production.

From a data science standpoint, knowing how to implement object-oriented programming is very valuable. Even if one is not a software engineer and does not typically maintain libraries or production considerations for a program — knowing when object-oriented programming should be used makes it a lot easier to implement proper production and version control procedures down the line.

Conclusion

In this article, you have seen:

- Why polymorphism is important in computer programming
- How to use polymorphism to incorporate functions under a class with a single name

- Calculation of probabilities using polymorphism

Like many concepts in computer programming, object-oriented programming is concerned with efficiency. For instance, just as using incorrect data types can lead to inefficient use of memory — storing functions separately can make a program more difficult to maintain. Object-oriented programming aims to solve that problem.

Many thanks for your time, and any questions or feedback are greatly appreciated. You can find more of my data science content at michael-grogan.com.

References

- numpy.org: numpy.random.binomial
- numpy.org: numpy.random.pareto
- Programiz: Polymorphism in Python
- statisticshowto.com: What is the Pareto Distribution?

Disclaimer: This article is written on an "as is" basis and without warranty. It was written with the intention of providing an overview of data science concepts, and should not be interpreted as professional advice. The findings and interpretations in this article are those of the author and are not endorsed by or affiliated with any third-party mentioned in this article. The author has no relationship with any third parties mentioned in this article.

Data Science | Object Oriented | Python | Probability | Polymorphism