INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING IN PYTHON

Winfred Lam, CFA

In previous issues of *The Analyst*, we reflected on Python's increasing prevalence in the financial industry, providing coverage of a web scraping class taught by the Marquee Group. In this article, we briefly introduce one of the most fundamental programming paradigms: object-oriented programming (OOP). OOP in Python is versatile and easy to pick up. This article uses scenarios from the financial services industry as examples to walk through OOP at a high level.

What is OOP?

OOP is a programming paradigm in which code is structured around a collection of objects. Each object is an instance of some user-defined class that contains both data and functions. By packaging data and functionality in one class, code becomes more reusable and readable.

OOP is supported by several programming languages. In fact, some languages, like Java, are built entirely around OOP. In this article, we walk through two basic OOP concepts in Python: classes and instances and inheritance.

Classes in Python

In Python, OOP starts with the class definition. Essentially, a class can be thought of as a blueprint for creating an object. An object is just an instance of some class.

Consider a security that trades on the stock exchange. A security has the following characteristics

- Name
- Ticker
- Price
- Shares outstanding

Every security issued has the above attributes (and more). We seek ways to take advantage of each security's common structure when writing functionality around it.

For example, each individual stock is a security. In OOP terms, each stock object is an instance of the class Security. That is, each stock can be thought of as an object.

Using the above, we define our own class called Security.

```
class Security:
   def __init__(self, name, ticker, price, shares_outstanding):
      self.name = name
      self.ticker = ticker
      self.price = price
      self.shares_outstanding = shares_outstanding
```

Constructor

Class creation in python requires a constructor, shown in the def _init_(self) method. The constructor helps initialize values when an instance of the class is created.

Every copy of the Security class has four common attributes, denoted as "name," "ticker," "price," and "shares_oustanding."

The constructor here says that, for each instance of Security to be created, there needs to be a value provided for name, ticker, price, and shares_out-standing. Here, the term self that precedes each attribute indicates that it belongs to the instance.

And that's it! We have now defined a class called Security. Every time there is a newly created security, we can reuse the same code to create it.

Methods

Now consider that we want each security to have some basic functions. For example, let's say we use the Security class to create a common stock object, and we want the common stock to display its ticker and market capitalization.

In the Security class, we also define the methods print_ticker() and print_market_cap(), which make use of the fact that every Security must have a name, ticker, price, and shares_outstanding, to print/perform basic operations. The methods will work for every security because, by definition, they must have these attributes.

```
class Security:
   def __init__(self, name, ticker, price, shares_outstanding):
      self.name = name
      self.ticker = ticker
      self.price = price
      self.shares_outstanding = shares_outstanding

   def print_ticker(self):
      print("The ticker for ", self.name, " is ", self.ticker)

   def print_market_cap(self):
      print("The market cap of the stock is: $", self.shares_outstanding * self.price)
```

Let us now create two stocks, Microsoft Corp. and Apple Inc., using the newly defined Security class. After creating the two stock objects, we can use the two methods print_ticker() and print_market_cap() to show the respective data. (Note that the values used here are for illustrative purposes only and do not represent those of the actual security trading on the exchange.)

```
msft_stock = Security('Microsoft Corp', 'MSFT US Equity', 280, 5000000)
apple_stock = Security('Apple Inc', AAPL US Equity', 150, 2000000)

msft_stock.print_ticker()
msft_stock.print_market_cap()

apple_stock.print_ticker()
apple_stock.print_market_cap()
```

Output

```
The ticker for Microsoft Corp is MSFT US Equity
The market cap of the stock is: $ 1400000000
The ticker for Apple Inc is AAPL US Equity
The market cap of the stock is: $ 300000000
```

What if we now want to simplify the print ticker message for all securities? We want it to print: Name: X Ticker: Y

The code design around objects means that we know exactly where to look—in the Security class. Regardless of how many stocks we end up creating, the modification can be done in one line:

```
class Security:
    def __init__(self, name, ticker, price, shares_outstanding):
        self.name = name
        self.ticker = ticker
        self.price = price
        self.shares_outstanding = shares_outstanding

    def print_ticker(self):
        print("Name: ", self.name, " Ticker: ", self.ticker)

    def print_market_cap(self):
        print("The market cap of the stock is: $", self.shares_outstanding * self.price)
```

```
msft_stock.print_ticker()
apple_stock.print_ticker()
```

Output

```
Name: Microsoft Corp Ticker: MSFT US Equity
Name: Apple Inc Ticker:  AAPL US Equity
```

Inheritance

A fundamental principle of OOP is inheritance. At the rudimentary level, inheritance means creating a subclass based on an existing class, where the subclass "inherits" all the properties and behaviours of the parent class. Python supports multiple inheritances.

Going back to the Security example, common shares are not the only securities trading on the stock market. Consider exchange-traded funds (ETFs). They fall under the bucket of "securities" and share more or less all the four attributes defined for the common stock. We can use the same code from the Security class when creating an ETF object. For example, suppose we create the ETF "zsp" with the following code:

```
zsp = Security('BMO S&P 500 Index ETF', 'ZSP CN Equity', 45, 100000)
```

Through **inheritance**, we can instead define a new class called ETF as a subclass of Security to better reflect the nature of the security. One significant difference is the addition of a "nav" (net asset value) variable.

```
class Security:
    def __init__(self, name, ticker, price, shares_outstanding):
        self.name = name
        self.ticker = ticker
        self.price = price
        self.shares_outstanding = shares_outstanding

    def print_ticker(self):
        print("Name: ", self.name, " Ticker: ", self.ticker)

    def print_market_cap(self):
        print("The market cap of the stock is: $", self.shares_outstanding * self.price)


class ETF(Security):

    def __init__(self, name, ticker, price, shares_outstanding, nav):
        super().__init__(name, ticker, price, shares_outstanding)
        self.nav = nav

    def print_market_cap(self):
        print("The size of the ETF is: $", self.shares_outstanding * self.price)

    def print_nav(self):
        print("The NAV of the ETF is: $", self.nav)
```

In the constructor, we added a parameter called "nav" using the line self.nav = nav. The line super().__init__(self, name, ticker, price, shares_oustanding, nav) indicates that the subclass (ETF) is calling the __init__() method of its parent class (Security) without explicitly naming it.

When creating a new ETF object zsp, we must now add a parameter as an argument, which our constructor of the ETF class mandates.

Continuing with our example, to create an instance of the class ETF as "zsp" we must also provide a "nav" as an additional argument. Suppose the ETF nav matches its price perfectly, with a value of 45.

```
zsp = ETF('BMO S&P 500 Index ETF', 'ZSP CN Equity', 45, 100000, 45)
```

The new ETF object can access the methods the Security class possesses. Any instance of the ETF class can reuse all the code previously defined in the Security class. For example, the ETF class can use the method print_ticker() from the Security class despite not having the method defined in the ETF class body—because an ETF is a Security.

Redefining methods in the subclass

Reviewing our earlier code, we can see that the message from print_market_cap() would not be suitable for the ETF. While the computation method is the same (price * shares outstanding), we do not want the message to say, "The market cap of the stock is: $xxx" for an ETF."

A more appropriate output would be: "The fund size of the ETF is: $xxx". So, in the subclass ETF, we redefine (or override) the print_market_cap() method to reflect a more suitable message.

In Python, this can be done by simply adding a method to the child class with the same name as in the parent class. In this case, we add the print_market_cap() method to the ETF class but modify its contents to print "fund size" and "ETF" instead of "market cap" and "stock," despite the respective methods from the parent and child class sharing the same calculation methodology. Now, when we call print_ticker() and print_market_cap(), the output more accurately reflects the nature of the security.

```
zsp.print_ticker()
zsp.print_market_cap()
```

Output

```
Name:  BMO S&P 500 Index ETF Ticker:  ZSP CN Equity
The fund size of the ETF is: $ 4500000
```

Adding methods to the subclass

Finally, we also add a method to the ETF class called print_nav(), which is not present in the Security class. This allows us to print the "nav" of every ETF object, a datapoint that was not part of its parent class Security.

As such, we see that the child class possesses all the attributes and methods of the parent class but can also redefine, or even add, some of its own. That is, a subclass can reuse all the parent class code while also adding or modifying functionality as needed. This is inheritance in a nutshell.

Conclusion

The above exercise introduced two fundamental OOP concepts, class and inheritance, but does not even scratch the surface of the power of OOP.

OOP requires careful planning and consideration of program structures at the beginning of the process. However, the payoff in the longer term is code reusability, improved organization, and better data structures, all of which can help save maintenance time and facilitate debugging as programs become more complex.

Winfred Lam, CFA, is a Senior Product Manager at BMO Global Asset Management and a graduate student in computer science at the University of Pennsylvania. He is a volunteer member of CFA Society Toronto's Member Communication Committee.

Engaging with companies on material ESG factors to drive change

To find out more, contact:

Natalie Bisaillon, MSc, CFA
Vice-President, Business Development and Client Relations
Natalie.Bisaillon@desjardins.com -
Tel 1-877-353-8686 ext. 5554450

Desjardins
Global Asset Management