TECHNOLOGY NEWS

HASHING IN PYTHON

Winfred Lam, CFA

Share this article

In the December 2022 edition of *The Analyst*, we provided a high-level introduction to object-oriented programming (OOP), which forms the backbone of the Python programming language. Having covered the basics of OOP and the concept of an "object," we will now move on to a more advanced topic: hashing. This critical concept goes hand in hand with objects in Python (and other OOP-centric languages). We will also explore the idea of equality in Python.

Hashing in Python is best illustrated in conjunction with data structures. As such, this article will broadly cover:

1. Data structures
2. Hashing
3. Equality

Compared to introduction to object-oriented programming In Python, this is a significant step up in difficulty and assumes basic comfort with Python code and OOP. As in the previous article, explanations of programming concepts will be supported with relatable examples from the financial services industry.

Data Structures in Python

In Python, users can create objects from their self-defined classes. We often write Python code using more ubiquitous, built-in data structures such as Integers, Strings, Lists, etc. These variables are also objects. In fact, every data structure in Python is an object.

There are two main categories of data structures in Python: primitive and non-primitive. Primitive data structures (also called data types) are, for lack of a better word, simpler. Non-primitive data structures are more advanced, as they are a collection of data types. Non-primitive data structures can store primitive data types and other non-primitive data structures. There are, however, rules for storing the latter, which will be the centrepiece of this article.

In Python, there are four primitive data types:

- Integers: Whole numbers from negative infinity to infinity (e.g., -1, 2, 10)
- Float: Floating point number, usually ending with decimal figures (e.g., 1.2, 2.0, 10.3333, 50000.123)
- Strings: A collection of alphabets. In Python, a string is created by enclosing some sequence of alphabets with ' or " (e.g., 'python', 'finance', "programming")
- Boolean: Only takes in the values True and False, interchangeable with the integers 1 and 0, essential to conditional and comparison expressions

On the other hand, we have non-primitive data structures:

- Array: Stores a collection of homogeneous data types
- List: Stores a collection of heterogeneous data structures
- Dictionary (also known as Map): Stores data in key:value pairs
- Set: Stores a collection of unordered, nonduplicate (unique) objects
- Tuple: An immutable (unchangeable) sequence of objects

Our discussion is chiefly concerned with non-primitive data structures. Typically, when people talk about Arrays in Python, they refer to Lists, as Arrays are less prevalent in Python than in other languages like Java, C, and C++. While there are significant fundamental differences between Arrays and Lists, for the purposes of this article, we will focus on List, Dictionary, Set, and Tuple.

All of four data structures (List, Dictionary, Set, and Tuple) can store primitive data structures. Non-primitive data structures can also store objects from user-defined classes. Bringing back the "Securities" example from the previous article, "Introduction to Object-Oriented Programming in Python," let us define a Securities class, instantiate several Securities objects, and add them to the above-mentioned data structures. We are also **overriding** the **str()** and **repr()** methods so that the objects will show the "Name" of each Security object when we invoke the **print()** method.

Note: The numbers used are hypothetical and do not represent those of the securities trading on the stock exchange.

Figure 1:

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

    def __str__(self):
        return self.name

    def __repr__(self):
        return self.name

msft_stock = Security('Microsoft Corp', 'MSFT US Equity', 300, 20000000)
apple_stock = Security('Apple Inc', 'AAPL US Equity', 150, 2000000)
zsp_etf = Security('BMO S&P 500 Index ETF', 'ZSP CN Equity', 45, 100000)
```

Figure 2:

```
[ ] #Creating a list with two securities in them
    #Then adding another one subsequently

    security_list = [msft_stock, apple_stock]
    security_list.append(zsp_etf)

    #Printing. Notice that the ordering is the same as the order in which they were added.
    for security in security_list:
        print(security)
```

Microsoft Corp
Apple Inc
BMO S&P 500 Index ETF

Figure 3:

```
[ ]  #Adding the securities to a set
     security_set = set()

     for element in security_list:
         security_set.add(element)

     # Notice that ordering is not the same for the set compared to the list, because a set is unordered.
     for security in security_set:
         print(security)
```

Apple Inc
BMO S&P 500 Index ETF
Microsoft Corp

Figure 4:

```
[ ]  #Adding securities to a dictionary.
     #The curly brackets {} indicate that security_headquarters is a dictionary.
     #Suppose we want to map the city location of the headquarters of each company to its stock

     security_headquarters = {'Redmond': msft_stock,
                              'Cupertino': apple_stock,
                              'Toronto': zsp_etf}

     print(security_headquarters)
     print(security_headquarters['Toronto'])
```

{'Redmond': Microsoft Corp, 'Cupertino': Apple Inc, 'Toronto': BMO S&P 500 Index ETF}
BMO S&P 500 Index ETF

Figure 5:

```
[ ]  #If a key is duplicated (e.g. Toronto), the value of the new occurence will override that of the prior entry
     #In this case, 'cash', a string, has replaced zsp_etf

     security_headquarters['Toronto'] = 'cash'
     print(security_headquarters)
     print(security_headquarters['Toronto'])
```

{'Redmond': Microsoft Corp, 'Cupertino': Apple Inc, 'Toronto': 'cash'}
cash

Figure 6:

```
[ ]  #We can also use the custom objects as the key, although in this case it would not make a lot of sense for our dictionary!

     security_headquarters[zsp_etf] = apple_stock
     print(security_headquarters)
```

{'Redmond': Microsoft Corp, 'Cupertino': Apple Inc, 'Toronto': 'cash', BMO S&P 500 Index ETF: Apple Inc]

The Unhashable TypeError

What if we tried using a List of locations as a Dictionary key for some custom ETF? Python will give us a TypeError saying the List is "unhashable":

Figure 7:

```
[ ]  #list as dictionary key
     custom_etf = Security('Custom ETF','CUST CN Equity',10, 1000)

     locations_list = ['Redmond','Cuptertino','Toronto']

     security_headquarters[locations_list] = custom_etf
```

```
TypeError Traceback (most recent call last)

<ipython-input-7-aece259ba1b7> in <module>
      4 locations_list = ['Redmond','Cuptertino','Toronto']
      5
----> 6 security_headquarters[locations_list] = custom_etf

TypeError: unhashable type: 'list'
```

Alternatively, what if we added a List to a Set? We will get the same error.

Figure 8:

```
[ ]  set_of_stocks = set()
     security_list.append(custom_etf)
     set_of_stocks.add(security_list)
```

```
TypeError Traceback (most recent call last)

<ipython-input-8-f5c9ee62c8a8> in <module>
      1 set_of_stocks = set()
      2 security_list.append(custom_etf)
----> 3 set_of_stocks.add(security_list)


TypeError: unhashable type: 'list'
```

Hashable

Every List, Dictionary, or Set we create is just an object. In Python, an object is unhashable if the object does not have a hash value. But what is a hash value? What is hashing?

Hashing is converting some key X into another value Y. This is done through a **hash function**, which helps generate the value Y via some mathematical algorithm. The value Y is known as a **hash value**, also known simply as **hash**. Generally, the input becomes a sequence of fixed length comprised of integers and/or alphabets.

The easiest example to explain this process is the creation of a password. When we create a password to login to a website, the password is not stored word for word on the website's database. If it was, it would not be a very good security system, as accounts would be easily compromised in the event that the website gets hacked, or if the database staff simply decided they wanted to have a peek at your account information. Instead, the password is hashed to produce a seemingly random, unreadable sequence that is stored on the server. This way, if the database does get compromised, the hacker (or staff) obtains the hash rather than your password.

Python has a standard library called hashlib, including some known hashing algorithms. Below are examples of hashes produced with the MD5 hash function **md5()**:

Figure 9:

```
[ ]  import hashlib

     #show hash functions available
     print(hashlib.algorithms_available)
     print()

     #md5
     hash_val = hashlib.md5(b"CFA")
     print(hash_val.hexdigest(), 'is the hash for CFA')

     #Small changes matter a lot. By changing one letter of CFA to lowercase, we get a completely different hash sequence
     hash_val2 = hashlib.md5(b"cFA")
     print(hash_val2.hexdigest(), 'is the hash for cFA')

     #Notice that increasing the length of the input string produces a hash of the same length
     hash_val3 = hashlib.md5(b"CFA Society")
     print(hash_val3.hexdigest(), 'is the hash for CFA Society')
```

{'md5-sha1', 'sha3_384', 'sha512_256', 'md4', 'sha256', 'blake2b', 'ripemd160', 'sha3_224', 'shake_128', 'sha1', 'blake2s', 'whirlpool', 'sha512', 'sha512_224', 'sm3', 'sha3_256', 'sha384', 'sha3_512', 'shake_256', 'sha224', 'md5'}

7d3e3deb50c19e7a3cafc081d9b3f23b is the hash for CFA
e8dd9b0a634a9e4d666947ed8daed46c is the hash for cFA
59018d5a5d867704f9bfdf7e181b8cbf is the hash for CFA Society

For a hash function to be effective in password protection, it has to fulfill certain conditions:

- For a given input, it must produce the same hash **every time**. Otherwise, you would type in the correct password and be unable to log in because the hash did not match the one in the database.
- It must be a one-way function (i.e., not reversible). Otherwise, a hacker could easily reverse the hash and obtain your password.
- As shown with MD5(), hashing is predicated on passing a String of any length through a hash function and producing an output of fixed length. The hash length must not change. This means that two passwords of different lengths produce hashes of the same length.

As such, even if a password leak happens and a cybercriminal obtains all the hashes from a database, a strong hash function can act as the ultimate line of defense, making it difficult for the cybercriminal to derive the password from the hash. The calculations required to reverse a hash to its original input are tough. Even if the cybercriminal can figure out what hash function was used, it is generally still faster to take a brute-force approach and feed millions of sequences into the hash function in the hope of producing a matching hash, a highly time-consuming process even for the strongest computers.

Sadly, your password is still unsafe. Seasoned cybercriminals are familiar with standard hash functions and can calculate hashes for known words. Weak and common passwords can be easily broken—a solid reminder not to set your password as "password."

Why Does This Matter in Data Structures?

Hashing has many applications, namely in cryptography and cybersecurity, and it is fundamental to data structures in programming. Essentially, a hash becomes a unique index for an element. Writing a good hash function is an art in itself

and beyond the scope of this article. Fortunately, Python has a built-in hash function that can be invoked via the **hash()** method. In Python, the essential properties of hashing are:

- The **hash()** method only returns a hash for immutable objects
- A **hashable object** is an object whose **hash value** never changes
- If two hashable objects compare as equal, they must have the same hash value

We can hash objects created from user-defined classes as well. In this case, let us produce the hash for the securities created. Unlike MD5(), the Python hash produces a sequence of integers.

Figure 10:

```
[ ]  print('msft_stock hash:',hash(msft_stock))
     print('apple_stock hash:',hash(apple_stock))
     print('zsp_etf hash:',hash(zsp_etf))
```

msft_stock hash: 8741861339679
apple_stock hash: 8741861339673
zsp_etf hash: 8741861339676

Immutable Objects

Every object in Python is classified as either mutable or immutable. A mutable object is an object whose state can be modified after it has been defined. An immutable object cannot be changed. This applies to both primitive and non-primitive data structures and objects created from user-defined classes.

Primitive data types like Integer, Float, String, and Boolean are immutable. Conversely, all the non-primitive data structures we just covered are mutable. We saw an earlier example of the "not hashable" error when we tried to use a List as a key for a Dictionary and also when we tried to add it to a Set. In comparison, the Tuple data structure is immutable and is heavily relied on as keys in Dictionaries when a sequence of numeric values is needed as a key.

The mutuable and immutable distinction is critical for data structures like Dictionaries and Sets. Even though they are not hashable, they still rely heavily on hashing, as it allows elements stored in these data structures to be indexed:

- In a Dictionary, for a key to reliably look up the mapped value, it must be able to discern between two keys. In the previous example, we saw that adding an entry with an existing key will overwrite the value from a previous entry. As such, the Dictionary must, without fail, recognize if a provided key is "the same" as an existing key. This is why a key for a Dictionary must be a hashable object.
- In a Set, the data structure needs to be able to judge whether a new element is a duplicate or not. That is, the Set must be able to verify whether an element is "the same" when compared to existing elements.

In both cases, the hashes of two objects are compared to make this judgment. This brings us to our last discussion, what does it mean for two objects to be equal?

Equality: Is GOOGL Equal to GOOG?

For primitive data types like Integers and Strings, equality is very easy to evaluate and appears not worthy of another thought. Obviously:

- 1 == 1 is True
- 'CFA' == 'CFA' is True
- 12.00 == 100.00 is False
- 'Microsoft' == 'Apple' is False

What about objects created from user-defined classes? In our example, how do we evaluate whether two Security objects are equal? Consider the below example:

Alphabet Inc., the parent company of Google, is publicly listed on the Nasdaq exchange. However, if you have ever looked at the top 10 stocks in the S&P 500 Index, you will see that there are two classes.

Figure 11:

Top 10 Constituents By Index Weight

CONSTITUENT	SYMBOL	SECTOR*
Apple Inc.	AAPL	Information Technology
Microsoft Corp	MSFT	Information Technology
Amazon.com Inc	AMZN	Consumer Discretionary
Nvidia Corp	NVDA	Information Technology
Tesla, Inc	TSLA	Consumer Discretionary
Berkshire Hathaway B	BRK.B	Financials
Alphabet Inc A	GOOGL	Communication Services
Alphabet Inc C	GOOG	Communication Services
Exxon Mobil Corp	XOM	Energy
Unitedhealth Group Inc	UNH	Health Care

*Based on GICS® sectors

Source: S&P Dow Jones Indices as of February 28, 2023

Both Alphabet Class A (GOOGL) and Alphabet Class C (GOOG) give shareholders ownership of the company. GOOGL and GOOG represent equal ownership, but GOOG does not provide voting rights and thus tends to trade at a slight discount. This distinction would matter for large shareholders or activist investors who want to influence management decisions in the company.

However, for smaller investors, this distinction might be less important. Regarding portfolio returns, GOOGL and GOOG effectively provide the same risk–reward profile, and an investor might only want to know whether or not they "own Google" in their portfolios. How is this implemented in Python code? Suppose we create the two securities:

Figure 12:

```
[ ]  google_stock_a = Security('Alphabet Inc', 'GOOGL US Equity', 90.1, 5000000)
     google_stock_c = Security('Alphabet Inc', 'GOOG US Equity', 90, 5000000)

     print(google_stock_a == google_stock_c)
```

False

From above, Python concludes that the two objects, google_stock_a and google_stock_c, are not equal, which is intuitively correct because the securities have different tickers and prices. If we check the hash of google_stock_a and google_stock_c, we will see that they differ.

Figure 13:

```
[ ]  print(hash(google_stock_a))
     print(hash(google_stock_c))
```

8741861339633
8741861339456

For our purposes, however, this can be problematic if we want to limit individual security weight to some threshold, but the program fails to conclude that GOOGL and GOOG are shares of the same company.

Let's say we add google_stock_a to our previously defined security_set. To track whether we already own a security, we rely on the Set's ability only to include non-duplicate elements to reject repeated additions. However, if we add google_stock_c, we will end up including Google twice because the Set thinks it is a different security, and we end up with more Google than we want in the portfolio.

Figure 14:

```
[ ]  security_set.add(google_stock_a)

     security_set.add(google_stock_c)

     for security in security_set:
         print(security)
```

Alphabet Inc
Alphabet Inc
Apple Inc
BMO S&P 500 Index ETF
Microsoft Corp

To let the program know that we want to treat all securities named "Alphabet Inc" as the same security, we need to **redefine (override)** what it means for two objects of the Security class to be equal. With regard to distinguishing the two in a Set, it is enough to just override the **hash()** method and not override the **eq()** method. This is because two objects with the same hash do not necessarily have to be equal, but two equal objects MUST have the same hash. To make things less confusing, we will walk through the process starting with **eq()**.

In Python, the equals method is denoted by **eq()** in the constructor. Remember from "Introduction to Object-Oriented Programming in Python" that we can redefine (override) a method by adding a method with the same name. Going back to our Security class definition:

Figure 15:

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

         def __str__(self):
             return self.name

         def __repr__(self):
             return self.name

         def __eq__(self, other):
             return isinstance(other, Security) and self.name == other.name
```

In the **eq()** method, we return True (i.e., the objects are equal) only if:

1. The object2 compared against object1 is also an instance of Security
2. The name attribute of the object2 (which is a String) is equal to the name attribute of object1

In this case, we know google_stock_a and google_stock_c are both instances of the class Security. We also know they share the same "name" attribute, namely "Alphabet Inc." As seen below, the line **google_stock_a == google_stock_c** will return true, and we can use this Boolean expression for other conditional comparisons.

Figure 16:

```
[ ]  google_stock_a = Security('Alphabet Inc', 'GOOGL US Equity', 90.1, 5000000)
     google_stock_c = Security('Alphabet Inc', 'GOOG US Equity', 90, 5000000)

     print(google_stock_a == google_stock_c)
```

```
True
```

We will now run into a new problem. Because we redefined the **eq()** method but did not also redefine the **hash()** method, if we try adding the securities to a Set, we will run into yet another unhashable type error, this time for the Security class.

Figure 17:

```
[ ]  security_set.add(google_stock_a)
```

```
TypeError                                 Traceback (most recent call last)

<ipython-input-17-495301c272b8> in <module>
----> 1 security_set.add(google_stock_a)


TypeError: unhashable type: 'Security'
```

Recall one of the core rules for hashing in Python:

- If two hashable objects compare as equal, they must have the same hash value

As such, we must redefine (override) the hash method. As we defined two stocks to be of the same company if they share the same name, regardless of ticker or price, we will implement it accordingly in the hash method, where we return the hash of self.name.

As self.name is a String, we know both hashes of the same name will return the same hash. Thus, this completes the override.

Figure 18:

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

         def __str__(self):
             return self.name

         def __repr__(self):
             return self.name

         def __eq__(self, other):
             return isinstance(other, Security) and self.name == other.name

         def __hash__(self):
             return hash(self.name)
```

Figure 19:

```
[ ]  google_stock_a = Security('Alphabet Inc', 'GOOGL US Equity', 90.1, 5000000)
     google_stock_c = Security('Alphabet Inc', 'GOOG US Equity', 90, 5000000)

     print(google_stock_a == google_stock_c)
     print(hash(google_stock_a))
     print(hash(google_stock_c))
```

```
True
7249468942858777719
7249468942858777719
```

With the hash method override also implemented, we can now be certain that objects created from the Security class will compare as we intended. As we can see below, a Set can successfully reject "duplicates." We create a new Set **security_set2** and add both google_stock_a and google_stock_c to it, but the Set correctly rejects the second addition because it now recognizes that they are the same stock, leaving only google_stock_a in the Set. We know we will not buy more Google stocks than we wanted!

Figure 20:

```
[ ]  security_set2 = set()

     security_set2.add(google_stock_a)
     security_set2.add(google_stock_c)

     print("Set size:",len(security_set2))

     for security in security_set2:
         print(security, ',', security.ticker)
```

Set size: 1
Alphabet Inc, GOOGL US Equity

Conclusion

While hashing is a mathematical algorithm that often works behind the scenes, it is critical to OOP and programming. Hashing also plays a crucial role in the lives of billions of non-programmers as it is applied in everything from password verification, cryptography, search algorithms, and blockchain. This humble process of transforming a given string of arbitrary length into another string of fixed length contributes far more to our lives than we realize!

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.