List, Tuples and Dataclasses

January 23, 2023

1 List in Python

1.1 Introduction

List is defined when there is an enclosing comma-separated sequence of objects in square brackets [ ], i.e [1, 2, 3]. They can contain one data types in one list i.e [“soo”, “look”] or multiple data types, i.e [1, “soop”, True, 2.321].

List is know to have three properties: - mutable (changeable) - ordered - dynamic, can be appended or trimmed

1.1.1 List can be constructed as follows:

[ ]:
```
a_list = [1, 2, 3]
a_list
```

[ ]:
```
[1, 2, 3]
```

List can also be constructed using Object Oriented Programming understanding as follows:

[ ]:
```
name = "python"
name = list(name)
name
```

[ ]:
```
['p', 'y', 't', 'h', 'o', 'n']
```

[ ]:
```
vowel_tuple = ('a', 'e', 'i', 'o', 'u')
vowel_list = list(vowel_tuple)
vowel_list
```

[ ]:
```
['a', 'e', 'i', 'o', 'u']
```

1.1.2 List can be subset like this:

[ ]:
```
a = ['foo', 'bar', 'baz', 'qux', 'quux']
a[2]
```

[ ]:
```
'baz'
```

[ ]:
```
a[1:4]
```

[ ]:
```
['bar', 'baz', 'qux']
```

One thing need to take note of is that list index starts from 0, which is why a[0] = 'foo'.

[ ]:
```
a[:4]
```

[ ]:
```
['foo', 'bar', 'baz', 'qux']
```

[ ]:
```
a[-1:]
```

[ ]:
```
['quux']
```

[ ]:
```
a[-1]
```

[ ]:
```
'quux'
```

[ ]:
```
a[:-1]
```

[ ]:
```
['foo', 'bar', 'baz', 'qux']
```

1.1.3 You can modify list like this:

Inserting

[ ]:
```
a
```

[ ]:
```
['foo', 'bar', 'baz', 'qux', 'quux']
```

[ ]:
```
a[1] = "gut"
a
```

[ ]:
```
['foo', 'gut', 'baz', 'qux', 'quux']
```

[ ]:
```
a.append(1)
a
```

[ ]:
```
['foo', 'gut', 'baz', 'qux', 'quux', 1]
```

Inserting multiple data into a list

[ ]:
```
# change 'gut' to [0, 'pool', 1.2]
a[1:2] = [0, 'pool', 1.2]
a
```

[ ]:
```
['foo', 0, 'pool', 1.2, 'baz', 'qux', 'quux', 1]
```

[ ]:
```
a[0:0] = [1, 2, 3]
a
```

[ ]:
```
[1, 2, 3, 'foo', 0, 'pool', 1.2, 'baz', 'qux', 'quux', 1]
```

Removing

```
[ ]: a[0:5] = []
     a
```

```
[ ]: ['pool', 1.2, 'baz', 'qux', 'quux', 1]
```

```
[ ]: a.remove(1.2)
     a
```

```
[ ]: ['baz', 'qux', 'quux', 1]
```

```
[ ]: del a[0]
     a
```

```
[ ]: [1.2, 'baz', 'qux', 'quux', 1]
```

1.2 Advantages

1. Can store different data types
2. Lists are mutable (changeable)

1.3 Disadvantages

1. List cannot be used for key in dictionaries
2. There's a possibility of data being changed, so the data is not protected enough

1.4 Best Practice for List

1. Sorting the values using .sort().
2. Use list as a building block for the dataframe columns
3. List comprehension, using a for loop and conditional if and else statement to generate a list. This will be more understandable and neater when writing the code.

2 Tuples in Python

2.1 Introduction

Tuples have these properties: 1. Ordered: Because a tuple's components are in a specific sequence, the significance of their placement within the tuple is evident. When you need to store information that must be in a precise sequence, such as a list of dates or a list of names, this can be helpful. 2. Heterogenous: Tuples can store elements of several types, including lists, strings, and numbers. This gives them a versatile data format that may be applied to various scenarios. 3. Immutable: A tuple's elements cannot be changed once it has been generated, making it immutable. To hold data that shouldn't be updated, a tuple is therefore a better option than a list. 4. Performance: Due to its immutability and fixed size, tuples are typically faster and more memory-efficient than lists. 5. Multi-Assignment: Tuples have the ability to perform multiple assignments, making it simple to assign values to numerous variables at once.

Tuples are identical to lists in all respect, except for the following properties.

1. Uses ( ) instead [ ]
2. Immutable

2.1.1 Tuples can be constructed as follows (same as list):

[ ]:
```
a_tuple = ("prot", "bar", "juily")
a_tuple
```

[ ]:
```
('prot', 'bar', 'juily')
```

Tuple can also be constructed using Object Oriented Programming understanding as follows:

[ ]:
```
name = "python"
name = tuple(name)
name
```

[ ]:
```
('p', 'y', 't', 'h', 'o', 'n')
```

[ ]:
```
vowel_tuple = ('a', 'e', 'i', 'o', 'u')
vowel_list = tuple(vowel_tuple)
vowel_list
```

[ ]:
```
('a', 'e', 'i', 'o', 'u')
```

2.1.2 Tuples can be subset like this (same as list) :

[ ]:
```
a = ('foo', 'bar', 'baz', 'qux', 'quux')
a[2]
```

[ ]:
```
'baz'
```

[ ]:
```
a[1:4]
```

[ ]:
```
('bar', 'baz', 'qux')
```

One thing need to take note of is that tuple index starts from 0, which is why a[0] = 'foo'.

[ ]:
```
a[:4]
```

[ ]:
```
('foo', 'bar', 'baz', 'qux')
```

[ ]:
```
a[-1:]
```

[ ]:
```
('quux',)
```

[ ]:
```
a[-1]
```

[ ]:
```
'quux'
```

```
[ ]: a[:-1]
```

```
[ ]: ('foo', 'bar', 'baz', 'qux')
```

2.1.3 You cannot modify tuples like list :

Because tuple is immutable, you will get an error for modifying their elements. For example,

```
[ ]: a
```

```
[ ]: ('foo', 'bar', 'baz', 'qux', 'quux')
```

```
[ ]: a[1] = "gut"
a
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/var/folders/nq/n31wszd53bs2y5jbzmryx3q00000gn/T/ipykernel_17058/697491499.py i
 ↪<module>
----> 1 a[1] = "gut"
      2 a

TypeError: 'tuple' object does not support item assignment
```

However, we can concatenate tuple just like list.

```
[ ]: ab = ("pops", "balloon")
a + ab
```

```
[ ]: ('foo', 'bar', 'baz', 'qux', 'quux', 'pops', 'balloon')
```

2.2 Advantages

1. Can store different data types unlike arrays
2. Tuple is immutable (unchangeable) which will be suitable for a data that need to be consistent

2.3 Disadvantages

1. Once a tuple have been created, you cannot modify the tuple anymore, so its not dynamic
2. Tuple are less functional than list and other data structures, making it less common in use

2.4 Best Practice for Tuples

1. To build intricate data structures with nested elements, stay away from using tuples. Use other data structures, like lists or dictionaries, for this instead.
2. Use the built-in comparison operators, like $<$ and $>$, when comparing tuples. The elements of the tuples are compared by these operators in lexicographic order, which means that the first element is compared first, followed by the second, and so on.¶

3. To represent a group of connected data objects with a set number of elements, use tuples. A tuple, for instance, can be used to symbolize a point in three dimensions, with each element standing in for the x, y, and z coordinates of the point.

3 Dataclasses in Python

This tutorial assumes that you already know the basics of object-oriented programming in general and have a working knowledge of Python. If you are not familiar with object-oriented programming, you should read the tutorial on Python classes first.

Useful links: https://realpython.com/python3-object-oriented-programming/

3.1 What are dataclasses?

Dataclasses are a way to create classes that are lightweight and have a lot of the functionality of a class without the overhead of a class. They are useful for creating classes that are used to store data. They are also useful for creating classes that are used to store data that is used to create other classes. The dataclasses are available through built-in dataclasses module in Python 3.7+.

[ ]:
```python
from dataclasses import dataclass
```

Let us create a dataclass that represents a bond. This can be achieved as follows:

[ ]:
```python
@dataclass
class Bond:
  """
  The dataclass that represents a bond.

  ## Parameters:
  rate: The rate of interest on the bond.
  duration: The duration of the bond.
  face_value: The face value of the bond.

  ## Properties:
  price: The present value of the future cash flows.
  """
  rate: float
  duration: float
  face_value: float

  @property
  def price(self) -> float:
    return self.face_value / (1 + self.rate) ** self.duration
```

The property decorator is used to store a computed attribute within a dataclass. The same code logic as above but without the dataclass decorator.

[ ]:
```python
class Bond:
  def __init__(self) -> None:
```

```
    self.rate = 0.0
    self.duration = 0.0
    self.face_value = 0.0

  def CalculatePresentValue(self) -> None:
    self.price = self.face_value / (1 + self.rate) ** self.duration
```

3.2 Why should you care about dataclasses when you have simpler tabular datastructures like Pandas dataframes?

One of the most powerful features with the Python dataclasses are that they support many features of object oriented programming. This would become clearer with the example of mortgages below. Mortgages are also a subclass of bonds, therefore we will simply inherit Bonds into a new dataclass called mortgages.

[ ]:
```
@dataclass
class Mortgage (Bond):
  pass
```

Now you can probably see that we have a structure. We have implemented programitacally that a mortgage is actually a bond and it can also have additional features. Mortgages will also have attributes that are specific to mortgages. We will add the down payment and the monthly payment to the mortgage class. Let us add those features and define the mortgage class again.

[ ]:
```
@dataclass
class Mortgage(Bond):
  down_payment: float
  @property
  def monthly_payment(self) -> float:
    monthly_rate = self.rate / 12
    return (self.face_value - self.down_payment) * monthly_rate / (1 - (1 +␣
 ↪monthly_rate) ** (-self.duration * 12))
  # We also need to override the price property. The price of the mortgage is␣
 ↪the present value of the monthly payments plus the down payment. This is␣
 ↪equal to the face value or the amount borrowed.
  @property
  def price(self):
    return self.face_value
```

Now let us create a mortgage object. Note that we need to supply 4 parameters: the rate, duration, face value and down payment. Three of these parameters are inherited from the Bond class. The down payment is a new parameter that is specific to the Mortgage class. Mortgage class also has an additional property attribute called monthly payment.

[ ]:
```
fixed_mortgage = Mortgage(rate=0.05, duration=30, face_value=100000,␣
 ↪down_payment=20000)
fixed_mortgage.monthly_payment
```

[ ]:
```
429.4572984097118
```

But mortgages can either be fixed rate or floating rate. We will now inherit from mortgage and create a floating rate mortgage. Now you could be probably getting a feel of how powerful dataclasses could be.

[ ]:
```
@dataclass
class FloatingRateMortgage(Mortgage):
  @property
  def monthly_payment(self) -> float:
      if(len(self.rate) != self.duration * 12):
          raise ValueError("The number of monthly rates is not equal to the␣
 ↪number of months in the mortgage duration.")
      # Now we will have a list of monthly payments. We will have to calculate␣
 ↪the monthly payment for each month based on the monthly rate.
      return [(self.face_value - self.down_payment) * (rate/12) / (1 - (1 +␣
 ↪(rate/12)) ** (-self.duration * 12)) for rate in self.rate]
```

We will now create a floating rate mortgage object. We will assume that the monthly annualised rate of interest is uniformly distributed between 4% and 12%. Note that the rate of interest is annualised and this annualised rate changes every month.

[ ]:
```
from numpy.random import uniform
floating_mortgage = FloatingRateMortgage(duration=30, face_value=100000,␣
 ↪down_payment=20000, rate=uniform(0.04, 0.12, 30*12))
floating_mortgage.monthly_payment[0:12] # We will print the first year of␣
 ↪monthly payments for sake of brevity.
```

[ ]:
```
[530.9941898888909,
 775.2555352740925,
 439.5470626720084,
 761.7033410945966,
 443.19156205169156,
 610.4539485787051,
 475.38375837483676,
 595.064992373187,
 507.8541106689975,
 471.5091034239094,
 547.3519518483176,
 390.21903287103135]
```

In the above example, we assumed that the rate of interest is uniformly distributed between 4% and 12%. This is quite unrealistic, but that is not the point. The point is to show, how powerful and convenient datatclasses can be. We used dataclasses to create a class (eg: Mortgage) that inherits from another class (eg: Bond) and add additional attributes and methods to the inherited class. By the way, you can also use your own interest rates or source the list of interest rates from any API such as FRED. You can also extend this code to create a useful application that would simulate different interest rate environments and show you how the mortgage payments would change or maybe something else (imagination is the limit!). That is the beauty of Python! You

can do powerful things with simple and readable tools.

Now let us look at other features that dataclasses provide. Suppose you are a small bank and you have given out 5 housing loans, with amounts of $10000, $50000, $60000, $90000 and $100000 respectively. All of these loans have a fixed interest rate of 5% and and a fixed duration of 30 years. The down payment for each loan is 20% of the loan amount.

[ ]:
```
loans = [10000, 50000, 60000, 90000, 100000]
mortgage_portfolio = [Mortgage(rate=0.05, duration=30, face_value=face_value,␣
 ↪down_payment=0.2 * face_value) for face_value in loans]
mortgage_portfolio
```

[ ]:
```
[Mortgage(rate=0.05, duration=30, face_value=10000, down_payment=2000.0),
 Mortgage(rate=0.05, duration=30, face_value=50000, down_payment=10000.0),
 Mortgage(rate=0.05, duration=30, face_value=60000, down_payment=12000.0),
 Mortgage(rate=0.05, duration=30, face_value=90000, down_payment=18000.0),
 Mortgage(rate=0.05, duration=30, face_value=100000, down_payment=20000.0)]
```

3.3 Data Wrangling with dataclasses

We have created a list of mortgage objects. Suppose we want to filter out the mortgages that have a face value greater than $50,000. Filtering the list of dataclasses is same as filtering any other list.

[ ]:
```
filtered_portfolio = list(filter(lambda loan: loan.face_value > 50000,␣
 ↪mortgage_portfolio))
filtered_portfolio
```

[ ]:
```
[Mortgage(rate=0.05, duration=30, face_value=60000, down_payment=12000.0),
 Mortgage(rate=0.05, duration=30, face_value=90000, down_payment=18000.0),
 Mortgage(rate=0.05, duration=30, face_value=100000, down_payment=20000.0)]
```

Or maybe you want to filter only those portfolios that have a monthly payment greater than $100

[ ]:
```
filtered_portfolio = list(filter(lambda loan: loan.monthly_payment > 100,␣
 ↪mortgage_portfolio))
filtered_portfolio
```

[ ]:
```
[Mortgage(rate=0.05, duration=30, face_value=50000, down_payment=10000.0),
 Mortgage(rate=0.05, duration=30, face_value=60000, down_payment=12000.0),
 Mortgage(rate=0.05, duration=30, face_value=90000, down_payment=18000.0),
 Mortgage(rate=0.05, duration=30, face_value=100000, down_payment=20000.0)]
```

[ ]:
```
# We can also sort dataclasses using the sorted function. We will sort the␣
 ↪mortgage portfolio based on the monthly payment in descending order.
sorted_portfolio = sorted(mortgage_portfolio, key=lambda loan: loan.
 ↪monthly_payment, reverse=True)
sorted_portfolio
```

```
[ ]: [Mortgage(rate=0.05, duration=30, face_value=100000, down_payment=20000.0),
      Mortgage(rate=0.05, duration=30, face_value=90000, down_payment=18000.0),
      Mortgage(rate=0.05, duration=30, face_value=60000, down_payment=12000.0),
      Mortgage(rate=0.05, duration=30, face_value=50000, down_payment=10000.0),
      Mortgage(rate=0.05, duration=30, face_value=10000, down_payment=2000.0)]
```

Anything that you can do with python lists, you can do with the list of dataclasses. For example, we can use the map function to add an attribute of amortization schedule to each mortgage object in the portfolio.

Let us define a function that will calculate the amortization schedule for a mortgage.

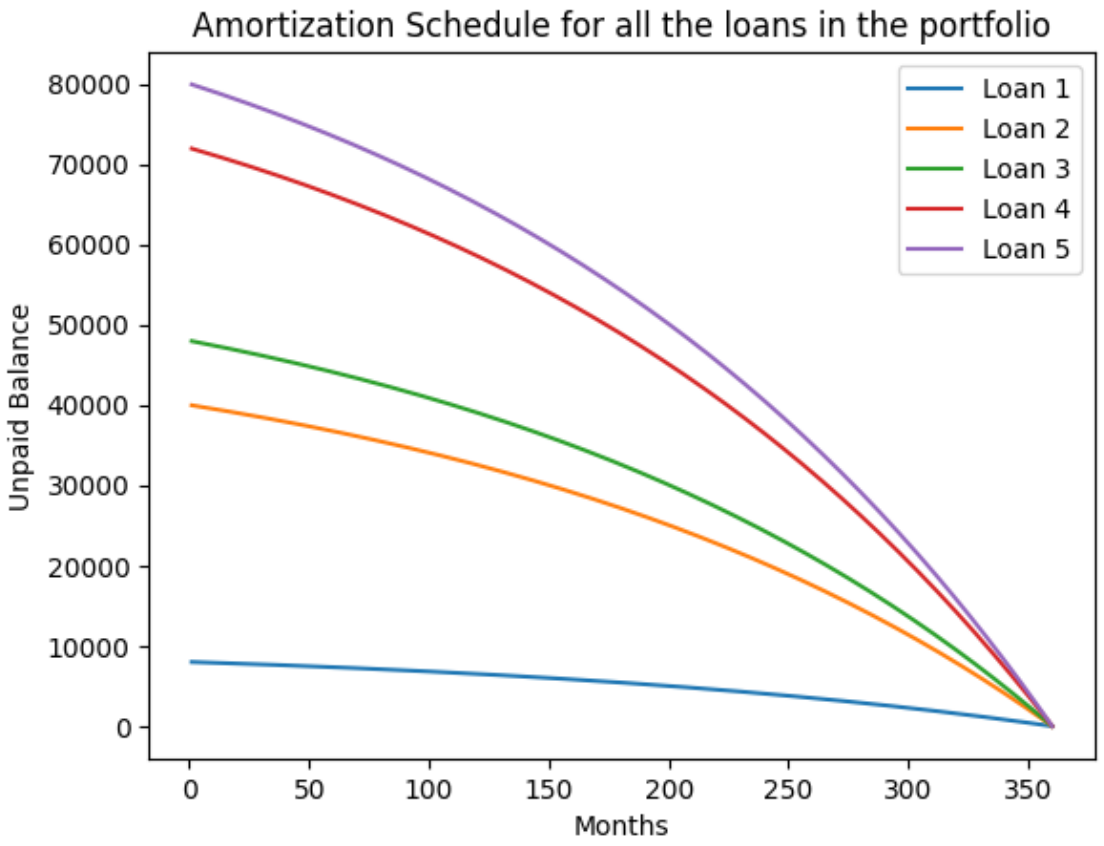
```
[ ]: def CalculateAmortizationSchedule(mortgage: Mortgage) -> list[dict]:
       monthly_rate = mortgage.rate / 12
       monthly_payment = mortgage.monthly_payment
       balance = mortgage.face_value - mortgage.down_payment
       amortization_schedule = []
       for i in range(0, mortgage.duration * 12):
         interest = balance * monthly_rate
         principal = monthly_payment - interest
         balance = balance - principal
         amortization_schedule.append({"period": i+1, "interest":interest,␣
      ↪"principal": principal, "unpaid_balance":balance})
       return amortization_schedule
```

Now let us use the map function to add the amortization schedule to each mortgage object in the portfolio.

```
[ ]: for mortgage in mortgage_portfolio:
       mortgage.amortization_schedule = CalculateAmortizationSchedule(mortgage)
     mortgage_portfolio[0].amortization_schedule[0:4] # We will print the first four␣
      ↪months of the amortization schedule for the first mortgage in the portfolio.
```

```
[ ]: [{'period': 1,
       'interest': 33.333333333333336,
       'principal': 9.612396507637854,
       'unpaid_balance': 7990.387603492362},
      {'period': 2,
       'interest': 33.293281681218176,
       'principal': 9.652448159753014,
       'unpaid_balance': 7980.73515533261},
      {'period': 3,
       'interest': 33.25306314721921,
       'principal': 9.692666693751981,
       'unpaid_balance': 7971.042488638858},
      {'period': 4,
       'interest': 33.21267703599524,
       'principal': 9.733052804975948,
       'unpaid_balance': 7961.309435833882}]
```

```
[ ]: import matplotlib.pyplot as plt
schedule_plots = [plt.plot([schedule["period"] for schedule in mortgage.
  ↪amortization_schedule], [schedule["unpaid_balance"] for schedule in mortgage.
  ↪amortization_schedule], label=f"Loan {i+1}") for i, mortgage in␣
  ↪enumerate(mortgage_portfolio)]
plt.title("Amortization Schedule for all the loans in the portfolio")
plt.xlabel("Months")
plt.ylabel("Unpaid Balance")
plt.legend()
plt.show()
```



3.4 Some disadvantages

There are multiple ways to achieve the same thing in Python. Dataclasses are useful but are not a substitute for pandas dataframes. Dataclasses do not have many out-of-the-box features as pandas dataframes provide such as rich support for date-time libraries and plotting. If the data makes more sense in a tabular format, then pandas dataframes are the way to go. For example, if you need time-series features like moving averages, differencing, etc., then do not use dataclasses, unless you are willing to write the code to implement these features.

However, dataclasses are useful when we want to create a simple data structure that is easy to

understand and maintain.