# Chapter 06 - Python Data Structures

April 4, 2021

## 0.1 Empty Objects

- we can create an object with out a subclass
- `o = object()`
- we cant allow an non-subclass object to have an attribute
- this is to save alot of memory
- when an object has an attribute, it takes system memory to keep track of it
- since every class uses object, If we add an atteribute to `object` and have thousands of classes, you now have that attribute stored thousands of times

- use **slots** to restrict arbitrary properties when you have classes that are duplicated thousands of time

[1]:
```
class MyObject:
    pass

m = MyObject()
m.x = "hello"
m.x
```

[1]:
```
'hello'
```

## 0.2 Tuples

- objects that can store a specific number of other objects in order
- they are `immutable`
- primary benefit of a tuple is that we can use them as keys in dictionaries
- we can also use them in other locations where objects require a hash value

- tuples only store data, behavior cannot be associated with a tuple
- the primary purpose of a tuple is to aggregate different pieces of data together into one container
- thus a tuple can be easiest tool to replace the object with no data
- we can use `slicing` to extract larger piecepieces of tuples

- major downside of a tuple is readability
- how does anyone know what the second item in the tuple is used for

### 0.3 Named Tuples

- what do we do when we want to group values together but we know we are frequently going to need to access them individually?
- we could use an empty object or dictionary
- first we have to import `namedtuple` then we have to add values as paramaters

[3]:
```
from collections import namedtuple

Stock = namedtuple("Stock", ["symbol", "current", "high", "low"])
stock = Stock("FB", 177.46, high=178.67, low=175.79)

stock
```

[3]:
```
Stock(symbol='FB', current=177.46, high=178.67, low=175.79)
```

[4]:
```
# we can unpack that tuple

print(stock.high)
```

```
178.67
```

### 0.4 Dataclasses

- Dataclasses are regular objects with a clean syntax for predefining attributes

[8]:
```
from dataclasses import make_dataclass

Stock = make_dataclass("Stock", ["symbol", "current", "high", "low"])
stock = Stock("FB", 177.46, high=178.67, low=175.79)

stock
```

[8]:
```
Stock(symbol='FB', current=177.46, high=178.67, low=175.79)
```

- obvious benefit of dataclass is that you can make it in 1 line and not 6
- you also have a much more useful string representation then the regular version
- it provides an equality comparision for free

[13]:
```
from dataclasses import dataclass
from typing import Any

@dataclass
class StockDecorated:
    name: str
    current: float = 0.0
    high: float = 0.0
    low: float = 0.0
    want: Any = "yes"
```

- with that we get comparisions for free
- we can set `order=True` and it will by default compare the values based on each of the attributes in the order they are defined
- you can customize the order by providing a `sort_index` attribute inside a `__post__init__` method on the class

## 0.5 Dictionaries

- dictionaries are increadibly useful containers that allow us to map objects directly to other objects
- attributes and properties of classes are stored internally as dictionaries
- dictionaries are objects that can hold other objects
- dictionaries have a `get` method
- a `setdefault` method to eithe return what was already there or the new thing that was created

[18]:
```
stocks = {
 "GOOG": (1235.20, 1242.54, 1231.06),
 "MSFT": (110.41, 110.45, 109.84),
}
stocks.setdefault("GOOG", "INVALID")
stocks.setdefault("BBRY", (10.87, 10.76, 10.90))
```

[18]:
```
(10.87, 10.76, 10.9)
```

- we can use `items` to iterate over dictionary key and values
- if you want ordered dictionary, use `OrderedDict`
- we cannot use a `list` as a key as it is not hashable
- objects that are `hashable` basically have a defined algorithm that converts the object into a unique integer value for rapid lookup
- tuples can be hashed because they are immutable
- lists can change and thus are unhashable
- simmilarly `dictionaries` cannot be used as keys

- we should use `dataclass` when we know exactly what attributes the data must store, espically if we also want to use the class definition as documentation for the end user
- `Dataclasses` might replace named tuples
- you cant loop over dataclases
- `dictionares` would be better choice if the keys describing the object are not known in advance, or if different objects will have some variety in their keys

- under the hood, most objects are implemented using dictionaries under the hood
- you can check this by looking at the `obj.__dict__` magic attribute
- when you use `obj.attr_name` it basically does the translation for you to `obj['attr_name']`

### 0.5.1 Using Defaultdict

- defaultdict comes with built in functions that handle alot for you
- you can use `defaultdict` to create containers
  - ‘graph = defaultdict(list)

### 0.5.2 Counter

- use it to count the instances of keys
- it has a `most_common()` attribute
- it returns a list of `(key, count)` from greatest to smallest

[22]:
```
from collections import Counter
responses = [
 "vanilla",
 "chocolate",
 "vanilla",
 "vanilla",
 "caramel",
 "strawberry",
 "vanilla"
]
print(
 "The children voted for {} ice cream".format(
 Counter(responses).most_common(1)[0][0]
 )
)
```

```
The children voted for vanilla ice cream
```

- `most_common(1)[0][0]`
  - you are requestion only 1 element
  - the element stores the name of the top element at position `0`
  - the key is the element `0`

## 0.6 Lists

- lists are the least object oriented data structures
- lists are used when we want to store several instances of the same type of object
- the benefit is that you get the order at which they were inserted, if that is the criteria
- we can modfiy a list how ever we see fit

- the question of lists is how we want to store it?
- do we want to store it as a `stack`, a `queue`, `linked list`, etc
- dont use lists for collecting different attributes of individual items

### 0.6.1 Sorting Lists

- if we want to place objects we define ourselves into a list and make those objects sortable, we have to do a bit more work
- the special `__it__` method, which stands for **less than** should be defined on the class to make instance of that class comparable

[31]:
```
from functools import total_ordering

@total_ordering
```

```python
class WeirdSortee:
    def __init__(self, string, number, sort_num):
        self.string = string
        self.number = number
        self.sort_num = sort_num

    def __lt__(self, object):
        if self.sort_num:
            return self.number < object.number
        return self.string < object.string

    def __repr__(self):
        return f"{self.string}:{self.number}"

    def __eq__(self, object):
        return all((
            self.string == object.string,
            self.number == object.number,
            self.sort_num == object.number
        ))


a = WeirdSortee('a', 4, True)
b = WeirdSortee('b', 3, True)
c = WeirdSortee('c', 2, True)
d = WeirdSortee('d', 1, True)

l = [a, b, c, d]

print("l")
print(l)
print("")
l.sort()
print(l)
```

```
l
[a:4, b:3, c:2, d:1]

[d:1, c:2, b:3, a:4]
```

- the sort function we have above can utilize duck-typing and as long as object has a `string`, `number` and `sort_num` attribute, it can sort it
- normally we would have to define `__gt__`, `__eq__`, `__ne__` but if we use the `@total_ordering` decorator, we can get them for free as long as we define `__it__` and `__eq__`
- normally, that above is overkill as we can just use `key` or `lambda`
- it is also common to sort a list of tuples by something other than the first item in the list, we can use `itemgetter`

[33]:
```
from operator import itemgetter

# we could honestly use lambda function for this
l = [('h', 4), ('n', 6), ('o', 5), ('p', 1), ('t', 3), ('y', 2)]
l.sort(key=itemgetter(1))
l
```

[33]:
```
[('p', 1), ('y', 2), ('t', 3), ('h', 4), ('o', 5), ('n', 6)]
```

### 0.7 Sets

- used for ensuring objects are unique
- sets can hold any `hashable` object
- sets are most useful when two or more of them are used in combination
- imagine using something like the `union` method
  - returns a new set with all elements that are in either of the two sets (| operator)
- there is also the `intersection` method
  - returns a new set that contains only those elements that are in both sets (& operator)
- there is also the `symmetric_difference` method that tells us what left
  - its the set of objects that are in one set or the other, but not both (^ operator)
- `union`, `intersection`, and `difference` can take in multiple sets
- there are also methods such as `issubset` and `issuperset` which return a boolean
- important to remember that unlike `union` and `intersection`, there is no symmetry
  - you cant do `item1.issubset(item2)` and get same results as `item2.issubset(item1)`
- `difference` returns all the elements that are in the **calling** set but not the **passed** set

### 0.8 Extending Built-in Functions

- refer to `python data models` to know more
- basically, you can use magic or dunder methods to override and extend built-in functions
- they usually start with double underscores
- to get the dunder methods all you have to do is type `dir(list)`

[36]:
```
dir(object)
```

[36]:
```
['__class__',
 '__delattr__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
```

```
'__lt__',
'__ne__',
'__new__',
'__reduce__',
'__reduce_ex__',
'__repr__',
'__setattr__',
'__sizeof__',
'__str__',
'__subclasshook__']
```

- if we need to somehow change any of the methods on the class, including speical methods, we definitly need to use inhertance
- if we used composition, we could write methods that perform the validation or alternations and ask the caller to use those methods
- dont try to extend