Assignment 3

Submission Date: January 2, 2018

- **The solutions to this assignment cannot contain any packages or material that was not taught during the lectures or recitations.**
- The assignment can be submitted in groups of two. If two partners work on the assignment, they should only upload one assignment to Moodle with the names and IDs of both participants as a comment in the solution file.
- Practical programming solutions should be submitted as `.py` files.
- You must implement the solutions given the function name provided.
- You are required to add `DocStrings` to all functions, as the first line of the body of the function.
- Submissions will be accepted only via Moodle!
- Any questions regarding the assignment should be directed to Dov Benyomin Sohacheski *(b@kloud.email)*.
- Extensions will be granted only by the lecturer.
- Late submission will not be graded and will result in a grade of zero.
- Functions should not receive user `input()` or `print()`. Rather, rely in function `args` and `return` values.
- All solutions should be submitted as a single file named `solutions.py`.

Part 1 -- Immutable Collections

You are required to create an *object-oriented collection* that will implement the *methods* described in this section. The object should be immutable, meaning that each method should return a new instance of the class and not alter the internal data. The code below should be used as boilerplate for your code.

Instance Methods

```
class Collection(object):
    pass
```

1. The `constructor` is defined as being able *(but not required)* to accept any iterable object except for `dict`: `__init__(self, iterable=None)`.

```
Collection()
Collection('hello world')
Collection(['hello', 'world'])
Collection({'hello', 'world'})
Collection(range(1000))
Collection(map(...))
```

2. The method `first(self)` should return a copy of the first item stored in the internal collection.
3. The method `last(self)` should return a copy of the last item stored in the internal collection.
4. The method `take(self, amount)` should return a new `Collection` contain a subset of the original items based on the amount desired. If the amount provided is greater than the total length of the collection, the additional amount should be ignored.

```
c = Collection([1,2,3,4])
print(c.take(2))
    # => Collection(1,2)
print(c.take(5))
    # => Collection(1,2,3,4)
```

5. The method `append(self, *elements)` should return a new `Collection` with the new elements appended to the end. Notice that `elements` is preceded with the `*` operator.

```
c = Collection([1,2,3,4])
print(c.append('hello'))
    # => Collection(1,2,3,4,'hello')
print(c.append(1,2,3,4))
    # => Collection(1,2,3,4,1,2,3,4)
print(c.append([1,2,3,4]))
    # => Collection(1,2,3,4,[1,2,3,4])
```

6. The method `prepend(self, *elements)` acts just like `append` but instead adds elements to the beginning of the new `Collection`.
7. The method `filter(self, *callbacks)` act just like the global method, however, it can receive many `callbacks` instead of just one. The method should return an new filtered `Collection`:

```
c = Collection([1,2,3,4,5,6,7,8])
print(c.filter(lambda number: number % 2, lambda number % 3))
    # => Collection(1,5,7)
```

8. The method `map(self, *callbacks)` act just like the global method, however, it can receive many `callbacks` instead of just one. The method should return a new mapped `Collection`:

```
c = Collection('hello')
print(c.map(str.upper, ord, chr))
    # => Collection('H','E','L','L','O')
```

9. The method `reduce(self, callback, initial=0)` act just like the global method. The method should return reduced value.

```
c = Collection([1,2,3,4])
print(c.reduce(add, 5))
    # => 15
print(c.reduce(add))
    # => 10
```

10. The method `sort(self, key=None, reversed=False)` should return a new sorted `Collection` based on the key provided. If no key was provided, the collection should be sorted using standard sorting strategies:

```
c = Collection('HELLO')
print(c.sort())
    # => Collection('E','H','L','L','O')
print(c.sort(True))
    # => Collection('O','L','L','H','E')
c = Collection([
    {'name': 'Joe', 'age': 20},
    {'name': 'Jane', 'age': 13}
])
print(c.sort(key='age'))
    # => Collection({'name': 'Jane', 'age': 13},{'name': 'Joe', 'age':
20})
```

11. The method `set(self, position, value)` should return a new `Collection` while setting the `value` at the `position` provided. If the position does not exist, no action should be taken and a copy of the `Collection` should be returned.

```
c = Collection('HELLO')
print(c.set(0, 'h'))
    # => Collection('h','E','L','L','O')
```

12. The method `pluck(self, key)` should return a new `Collection` with the only the `key` of each element. If the internal elements are not dictionaries, then no action should be taken and a copy of the current collection should be returned.

```
c = Collection([
    {'name': 'Joe', 'age': 20},
    {'name': 'Jane', 'age': 13}
])
print(c.pluck('age'))
    # => Collection(20,13)
print(c.pluck('name'))
    # => Collection('Joe','Jane')
```

13. The method `values(self)` should return a copy of the internal values.

```
c = Collection('HELLO')
print(c.values())
    # => ('H', 'E', 'L', 'L', 'O')
```

13. The method `unique(self)` should return a new `Collection` with only unique items.

```
c = Collection('HELLO')
print(c.unique())
    # => Collection('H','E','L','O')
```

14. The method `tap(self, callback)` should pass each element of the collection by-value to a callback function.

```
c = Collection('HELLO')
c.tap(print)
    # => 'H'
    # => 'E'
    # => 'L'
    # => 'L'
    # => 'O'
```

Special Methods

1. The method `__getitem__(self, position)` should return the item at a given `position`. If the position provided does not exist, `None` should be returned:

```
c = Collection('hello')
print(c[0])
    # => 'h'
print(c[-1])
    # => 'o'
print(c[17])
    # => None
```

2. The method `__add__(self, other)` should concatenate two collections or a collection with an `iterable` object.

```
c1 = Collection([1,2,3,4])
c2 = Collection([1,2,3,4])
iterable = 'hello'
print(c1 + c2 + iterable)
    # => Collection(1,2,3,4,1,2,3,4,'h','e','l','l','o')
```

3. The method `__sub__(self, other)` should return a new `Collection` containing items that exist in the first collection but not in the second. It can also be passed an `iterable` object.

```
c1 = Collection([1,2,3,4])
c2 = Collection([1,2])
iterable = 'hello'
print(c1 - c2)
    # => Collection(3,4)
print(c1 - iterable)
    # => Collection(1,2,3,4)
```

3. The method `__len__(self)` should return the length of the `Collection`.

```
c1 = Collection([1,2,3,4])
c2 = Collection([1,2])
print(len(c1))
    # => 4
print(len(c2))
    # => 2
```

4. The method `__contains__(self, element)` should return the existence of an `element` in the `Collection`.

```
c1 = Collection([1,2,3,4])
c2 = Collection([1,2])
print(1 in c1)
    # => True
print(3 in c2)
    # => False
```

5. The method `__eq__(self, other)` should return the whether all the elements of the two `Collection`s are equal.

```
c1 = Collection([1,2,3,4])
c2 = Collection([1,2])
c3 = Collection([1,2,3,4])
print(c1 == c2)
    # => False
print(c1 == c2)
    # => True
```

6. The method `__ne__(self, other)` should be the negation of the equals operator.
7. The method `__str__(self)` should return a string representation of the elements of the object.

```
c1 = Collection([1,2,3,4])
c2 = Collection([1,2])
print(c1)
    # => (1,2,3,4)
print(c2)
    # => (1,2)
```

8. The method `__repr__(self)` should return a programatic representation of the elements of the object. *(As in the examples above)*

```
c1 = Collection([1,2,3,4])
c2 = Collection([1,2])
print(repr(c1)
    # => Collection(1,2,3,4)
print(c2)
    # => (1,2)
```

Part 2 -- Pipelines

The following section will utilize the methods implemented above. You are to build a `pipeline` for the question below while using data provided to you in the form of a `json` file. **The names of the files are critical and should not be changed.** Example datasets will be provided for each question on Moodle.

How to open a file

Assuming your data is stored in `data.json` as follows:

```
{ "office":
    {"medical": [
      { "room-number": 100,
        "use": "reception",
        "sq-ft": 50,
        "price": 75
      },
      { "room-number": 101,
        "use": "waiting",
        "sq-ft": 250,
        "price": 75
      },
      { "room-number": 102,
        "use": "examination",
        "sq-ft": 125,
        "price": 150
      },
      { "room-number": 103,
        "use": "examination",
```

```
        "sq-ft": 125,
        "price": 150
      },
      { "room-number": 104,
        "use": "office",
        "sq-ft": 150,
        "price": 100
      }
    ]},
    "parking": {
      "location": "premium",
      "style": "covered",
      "price": 750
    }
}
```

You can load the data with the snippet below.

```
import json

with open('data.json', 'r') as f:
    data = json.load(f)

print(data["office"]["parking"]["style"])
    # => "covered"
```

Task

The file `waze.json` contains a feed of the alerts posted by users in the last 24 hours. Your task is to create a pipeline that will implement the following actions on the collection. You are to create methods based on the boilerplate below and return a new collection at each step.

- **Enumerate:** load the dataset from the `waze.json` file.
- **Remove noise:** remove any invalid alerts. An alert is considered invalid if it is missing any of the following properties: `country`, `reliability`, or `user`.
- **Completing values:** alerts the do not have a `type` value should be given a `type='other'` by default.
- **Accumulate**
  - Get the average reliability for all alerts in `Israel`.
  - Create a collection of the top 100 most active `users` based on the amount of alerts they posted sorted from most popular to least.
  - Get the user who posts the most amount of accidents.
  - Get a collection of alert types and their counts by Country.

Example `waze.json`

```
[
    {"country": "US", "reliability": 6, "user": "lsteffans0", "type": "accident", "time": "16:10:36"},
    {"country": "IL", "reliability": 5, "user": "rdunton1", "type": "police", "time": "4:37:22"},
    {"country": "GR", "reliability": 1, "user": "kcardew2", "type": "accident", "time": "11:25:40"},
    {"country": "IL", "reliability": 10, "user": "ogooble3", "type": "closed", "time": "2:18:42"},
    {"country": "FR", "reliability": 3, "user": "lhasel4", "type": "police", "time": "12:14:20"},
    {"country": "US", "reliability": 2, "user": "lsteffans0", "type": "police", "time": "2:08:27"}
]
```

Boilerplate

```
filename = 'waze.json'
print(enumerate_waze(filename))
    # => Collection({"country": "US", "reliability": 6, "user": "lsteffans0"......)
c = enumerate_waze(filename)
c.tap(print)
    # => {"country": "US", "reliability": 6, "user": "lsteffans0"......

c = clean_noise(c)
c = complete_values(c)

israeli_reliability    = get_average_reliability(c)
print(israeli_reliability)
    # => 7.5

top_users = get_top_100_users(c)
print(top_users)
    # => Collection('lsteffans0',rdunton1,......)

top_accidents = get_top_accident_notifyer(c)
print(top_accidents)
    # => 'lsteffans0'

alerts_types = get_alert_types_by_country(c)
print(alerts_types)
    # => Collection({"country": "US", "data": Collection({"accident": 1, "police": 1})},{"country": "IL", "data":......)
```