lec-othercollections

December 8, 2016

1 Collection data types

Tomáš Svoboda, Petr Pošík Department of Cybernetics, FEE CTU in Prague EECS, BE5B33PRG: Programming Essentials, 2016

Requirements (we already know): * Strings, tuples, lists * Loops

Why? * Power * Spam filter task

2 Warm up a bit . . .

```
In [1]: from IPython.display import Image
        Image('symbols_collection.png')
```

```
Out[1]:
```

3 Collections we already know: Sequence types

- Sequences of items. They support
- membership operator `in`,
- querying for size (`len()`),
- indexing and slices (`[]`), and are
- iterable.
- **string**: *imutable ordered* sequence of characters
- **tuple**: *imutable ordered* sequence of items of any data type
- **list**: *mutable ordered* sequence of items of any data type

3.1 in - example

```
In [2]: a = [2,3,4]
        b = "234"
        c = (2,3,4)
```

3.2 slices [:]

```
In [3]: a[:]
        a[-2:] == [3,4]

Out[3]: [2, 3, 4]

Out[3]: True
```

4 Set types

- Set types support
- membership operator (`in`),
- querying for size (`len()`), and are
- iterable.
- They also support **set operations** (comparisons, union, intersection, subset).
- **set**: *mutable unordered* collection of *unique* items of any data type
- **frozenset**: *immutable unordered* collection of *unique* items of any data type

Set types

- When **iterated** over, set types provide their items **in an arbitrary order**.
- Only **hashable** objects may be added to a set.
- Immutable data types are hashable (int, float, str, tuple, frozenset).
- Mutable values are (usually) not hashable (list, dict, set), since their contents can change.

4.1 Basic set usage

Creating a set of letters from a sequence of letters:

```
In [4]: s = set('abracadabra')
        s

Out[4]: {'a', 'b', 'c', 'd', 'r'}
```

Iterating over set items:

```
In [5]: for i in s:
            print(i, end=' ')

a b r c d
```

Membership checking:

```
In [6]: 'a' in s, 'z' in s

Out[6]: (True, False)
```

Adding an item to a set:

```
In [7]: s.add('z')
        s

Out[7]: {'a', 'b', 'c', 'd', 'r', 'z'}
```

Removing an item from a set:

```
In [8]: s.discard('a')  # Nothing happens if 'a' not in s
        s

Out[8]: {'b', 'c', 'd', 'r', 'z'}

In [9]: s.add((2,3))   # Raises KeyError if 'b' not in s
        s

Out[9]: {'b', 'z', 'r', 'c', (2, 3), 'd'}
```

4.2 Set operations

```
In [10]: set('programming'), set('essentials')

Out[10]: ({'a', 'g', 'i', 'm', 'n', 'o', 'p', 'r'}, {'a', 'e', 'i', 'l', 'n', 's',
```

Union:

```
In [11]: set('programming') | set('essentials')

Out[11]: {'a', 'e', 'g', 'i', 'l', 'm', 'n', 'o', 'p', 'r', 's', 't'}
```

Intersection:

```
In [12]: set('programming') & set('essentials')

Out[12]: {'a', 'i', 'n'}
```

Difference:

```
In [13]: set('programming') - set('essentials')

Out[13]: {'g', 'm', 'o', 'p', 'r'}
```

Symmetric difference:

```
In [14]: set('programming') ^ set('essentials')

Out[14]: {'e', 'g', 'l', 'm', 'o', 'p', 'r', 's', 't'}
```

4.3 Set “comparisons”

Are two sets disjoint? (I.e., is their intersection empty?)

```
In [15]: set('programming').isdisjoint(set('essentials'))
```

```
Out[15]: False
```

Is one subset of another?

```
In [16]: set('pro') <= set('programming')  # Or, set('pro').issubset(set('programmi
```

```
Out[16]: True
```

Is one superset of another?

```
In [17]: set('pro') >= set('programming')  # Or, set('pro').issuperset(set('progran
```

```
Out[17]: False
```

4.4 Set example: unique items

Having a list of (e.g.) words, how do we get a list of unique words?

```
In [18]: words = 'one three two one two one'.split()
         print(words)
```

```
['one', 'three', 'two', 'one', 'two', 'one']
```

```
In [19]: unique_words = list(set(words))
         print(unique_words)
```

```
['three', 'one', 'two']
```

Note, however, that the new list does not (in general) preserve the order of words in the original list.

4.5 Set example: eliminate unwanted items (1)

Having a list of file names, how do we get rid of some of them (`!prediction.txt`, `!truth.txt`)?

```
In [20]: orig_filenames = 'f1 f2 !prediction.txt f3 f4.ext !truth.txt f5'.split()
         orig_filenames
```

```
Out[20]: ['f1', 'f2', '!prediction.txt', 'f3', 'f4.ext', '!truth.txt', 'f5']
```

```
In [21]: filenames = set(orig_filenames)
         print(filenames)
         for fname in {'!truth.txt', '!prediction.txt'}:
             filenames.discard(fname)
             print(filenames)
```

```
{'!prediction.txt', 'f4.ext', 'f3', 'f5', 'f2', '!truth.txt', 'f1'}
{'f4.ext', 'f3', 'f5', 'f2', '!truth.txt', 'f1'}
{'f4.ext', 'f3', 'f5', 'f2', 'f1'}
```

4.6 Set example: eliminate unwanted items (2)

Having a list of file names, how do we get rid of some of them (`!prediction.txt`, `!truth.txt`)?

```
In [22]: filenames = set(orig_filenames)
         print(filenames)
```

```
{'!prediction.txt', 'f4.ext', 'f3', 'f5', 'f2', '!truth.txt', 'f1'}
```

```
In [23]: filenames = filenames - {'!truth. txt', '!prediction.txt'}
         filenames
```

```
Out[23]: {'!truth.txt', 'f1', 'f2', 'f3', 'f4.ext', 'f5'}
```

5 getting filenames

```
In [24]: import os
         fnames = os.listdir('./data')
         print(fnames)
```

```
['2852-0.txt', 'pg1112.txt', 'pg1112.txt~', 'pg1524.txt']
```

5.1 Set example: word analysis

```
In [25]: with open('./data/pg1112.txt','rt',encoding='utf-8') as f:
             book1_text = f.read()
         with open('data/2852-0.txt','rt',encoding='utf-8') as f:
             book2_text = f.read()
```

```
In [26]: words1 = book1_text.split()
         words2 = book2_text.split()
         unique_words1 = set(words1)
         unique_words2 = set(words2)
```

```
In [27]: len(unique_words2)/len(words2)
```

```
Out[27]: 0.15836470701805194
```

6 Mapping types

- A mapping type is an **unordered collection of key-value pairs**. They support
- membership operator `in`,
- querying for size (`len()`), and are
- iterable.
- Only hashable (i.e. immutable) objects can be used as keys.
- Each key's associated value may be of any data type.

6.1 Dictionary

Creating a dictionary:

```
In [28]: course = {'id': 'BE5B33PRG', 'name': 'Programming essentials', 'capacity':
         course2 = dict(id='BE5B33PRG', name='Programming essentials', capacity=25)
         course3 = dict([('id', 'BE5B33PRG'), ('name', 'Programming essentials'),
         course4 = dict(zip(('id', 'name', 'capacity'), ('BE5B33PRG', 'Programming
```

All the above methods create a dictionary with the same contents:

```
In [29]: course

Out[29]: {'capacity': 25, 'id': 'BE5B33PRG', 'name': 'Programming essentials'}

In [30]: course == course2 == course3 == course4

Out[30]: True
```

Testing membership in a dictionary (the tested object is assumed to be a key):

```
In [31]: 'id' in course, 'BE5B33PRG' in course

Out[31]: (True, False)
```

Querying a dictionary for a value:

```
In [32]: course['id']

Out[32]: 'BE5B33PRG'
```

Getting the lists of keys, values and key-value pairs:

```
In [33]: print(list(course.keys()))
         print(list(course.values()))
         print(list(course.items()))

['name', 'id', 'capacity']
['Programming essentials', 'BE5B33PRG', 25]
[('name', 'Programming essentials'), ('id', 'BE5B33PRG'), ('capacity', 25)]
```

Adding new key-value pairs:

```
In [34]: course['lecturer'] = 'Svoboda'
         print(course)
```

```
{'name': 'Programming essentials', 'lecturer': 'Svoboda', 'id': 'BE5B33PRG', 'capac
```

Replacing a value for an existing key:

```
In [35]: course['lecturer'] = 'Posik'
         print(course)
```

```
{'name': 'Programming essentials', 'lecturer': 'Posik', 'id': 'BE5B33PRG', 'capacit
```

Removing an item from a dictionary:

```
In [36]: del course['lecturer']
         print(course)
```

```
{'name': 'Programming essentials', 'id': 'BE5B33PRG', 'capacity': 25}
```

6.2 Iterating over dictionaries

Iterating over keys:

```
In [37]: for key in course:
             print(key, course[key], end=' | ')
```

```
name Programming essentials | id BE5B33PRG | capacity 25 |
```

or

```
In [38]: for key in course.keys():
             print(key, end=' | ')
```

```
name | id | capacity |
```

Iterating over values:

```
In [39]: for val in course.values():
             print(val, end=' | ')
```

```
Programming essentials | BE5B33PRG | 25 |
```

Iterating over key-value pairs:

```
In [40]: for item in course.items():
             print(item[0], '=', item[1], end=' | ')
```

```
name = Programming essentials | id = BE5B33PRG | capacity = 25 |
```

or, in a better way:

```
In [41]: for key, val in course.items():
             print(key, '=', val, end=' | ')
```

```
name = Programming essentials | id = BE5B33PRG | capacity = 25 |
```

6.3 `dict.get()` method

Returns * the value corresponding to the key, if the key exists in the dictionary, * `None` if key is not in the dictionary and no default value is given, or * a default value, if key does not exist in the dictionary and the default value is specified.

```
In [42]: print(course['id'])

BE5B33PRG
```

```
In [43]: print(course.get('id'))

BE5B33PRG
```

Querying a value for a non-existent key:

```
In [44]: course

Out[44]: {'capacity': 25, 'id': 'BE5B33PRG', 'name': 'Programming essentials'}

In [45]: #print(course['univ'])      # Raises KeyError

In [46]: print(course.get('univ'))

None
```

```
In [47]: print(course.get('id', 'CTU'))

BE5B33PRG
```

```
In [48]: print(course.get('univ', 'CTU in Prague'))

CTU in Prague
```

6.4 words analysis

```
In [49]: wf = {}
         for word in words1:
```

```
      File "<ipython-input-49-49f15779594d>", line 7

    ^
SyntaxError: unexpected EOF while parsing
```

7 Counter

- A special kind of a mapping type (dictionary).
- Collection of **elements** which are stored as keys, and their **counts** are stored as **values**.
- Values are counts, i.e. any integers, **including negative**.
- Defined in `collections` module:

```
from collections import Counter
```

7.1 Creating a Counter

```
In [ ]: from collections import Counter
        c = Counter()                          # a new, empty counter
        c = Counter('abracadabra')             # a new counter from an iterable
        c = Counter({'red': 4, 'blue': 2})     # a new counter from a mapping
        c = Counter(cats=4, dogs=8)            # a new counter from keyword args
```

7.2 Accessing Counter elements

- Use indexing as for `dict`s.
- For non-existing keys, `Counter` returns 0, instead of raising `KeyError`.

```
In [ ]: c = Counter(['eggs', 'ham'])
        print(c)
```

```
In [ ]: print(c['eggs'])
        print(c['bacon'])
```

7.3 `Counter.most_common()`

```
In [ ]: c = Counter('abracadabra')
        c
```

```
In [ ]: c.most_common(3)
```

7.4 Adding and subtracting counters

```
In [ ]: c1 = Counter('abracadabra')
        c2 = Counter('simsalabim')
        print(c1)
        print(c2)
```

```
In [ ]: print(c1 + c2)
```

```
In [ ]: print(c1 - c2)
```

Note, there are no elements with negative values (that could be expected for `s`, `i`, `m`, ...).

7.5 `Counter.update()` and `Counter.subtract()`

```
In [ ]: c = Counter()
        c1 = Counter('abrakadabra')
        c2 = Counter('avada kedavra')
```

```
In [ ]: c.subtract(c1)      # Negative counts
        print(c1)
        print(c)
```

```
In [ ]: c.update(c2)
        print(c)
```

```
In [ ]: c.update(c1)
        c.subtract(c2)
        print(c)
```

8 Named tuple

- **Named tuple is still a tuple.** You can use a named tuple everywhere you can use a tuple.
- It adds the ability to **refer to tuple items by names**, in addition to indexing by numbers.
- The closest relative to `struct` or `record` known from other programming languages.
- Usage:

```
from collections import namedtuple
# Create a custom tuple data type
Sale = namedtuple('Sale', 'customerid date productid quantity price')
# Create an instance of the new data type
sale = Sale(111, '2015-11-26', 222, 3, 2.50)
```

- Function `namedtuple` creates **a customized tuple data type**:
  - Arg 1: The name of the new data type.
  - Arg 2: String with space-separated names, one for each item in our customized tuple.

8.1 Named tuple: Example 1

```
In [ ]: from collections import namedtuple

        Sale = namedtuple('Sale', 'customerid date productid quantity price')
        sale = Sale(111, '2015-11-26', 222, 3, 2.50)

        # Now, you can access the individual items by indexing
        print(sale[1], sale[2], sale[3])
        print(sale[1:4])
        # ... or by names
        print(sale.date, sale.productid, sale.quantity)
```

```
In [ ]: # Create a bill consisting of several sales
        sales = [Sale(111, '2015-11-26', 222, 3, 2.50),
                 Sale(111, '2015-11-26', 231, 1, 7.50),
                 Sale(111, '2015-11-26', 12, 5, 3.00)]

        # Compute the total
        total = 0
        for sale in sales:
            total += sale.quantity * sale.price
        print('Total: ${:.2f}'.format(total))
```

8.2 Named tuple: Example 2

You can also nest one named tuple inside another, there is nothing special about it.

```
In [ ]: from collections import namedtuple

        Aircraft = namedtuple('Aircraft', 'manufacturer model seats')
        Seating = namedtuple('Seating', 'min max')

        aircraft = Aircraft('Airbus', 'A320-200', Seating(100, 220))
        print(aircraft)
        print(aircraft.seats.max)
```

8.3 Extracting items of named tuples for printing

```
In [ ]: print(aircraft[0], aircraft[1])
        print(aircraft.manufacturer, aircraft.model)
        print('{} {}'.format(aircraft.manufacturer, aircraft.model))
        print('{0.manufacturer} {0.model}'.format(aircraft))
        print('{manufacturer} {model}'.format(**aircraft._asdict()))
```

9 Summary

- The rich set of data collections is one of the main reasons for the **Batteries included!** Python slogan.
- Even more specialized data types can be created by using classes. See next lectures!

10 Notebook config

Some setup follows. Ignore it.

```
In [ ]: from notebook.services.config import ConfigManager
        cm = ConfigManager()
        cm.update('livereveal', {
                      'theme': 'Simple',
                      'transition': 'slide',
                      'start_slideshow_at': 'selected',
                      'width': 1268,
                      'height': 768,
                      'minScale': 1.0
        })
```

```
In [ ]: %%HTML
        <style>
        .reveal #notebook-container { width: 90% !important; }
        .CodeMirror { max-width: 100% !important; }
        pre, code, .CodeMirror-code, .reveal pre, .reveal code {
            font-family: "Consolas", "Source Code Pro", "Courier New", Courier, mor
        }
        pre, code, .CodeMirror-code {
            font-size: inherit !important;
        }
        .reveal .code_cell {
            font-size: 130% !important;
            line-height: 130% !important;
        }
        </style>
```