Collection data types

Petr Pošík

Department of Cybernetics, FEE CTU in Prague

EECS, BE5B33PRG: Programming Essentials, 2015

Requirements:

- Strings, tuples, lists
- Loops

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

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

- 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.

Basic set usage

Creating a set of letters from a sequence of letters:

In [1]:
```
s = set('abracadabra')
s
```

Out[1]:
```
{'a', 'b', 'c', 'd', 'r'}
```

Iterating over set items:

In [2]:
```
for i in s:
    print(i, end=' ')
```

```
d c a b r
```

Membership checking:

In [3]:
```
'a' in s, 'z' in s
```

Out[3]:
```
(True, False)
```

Adding an item to a set:

In [4]:
```
s.add('z')
s
```

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

Removing an item from a set:

In [5]:
```
s.discard('a')  # Nothing happens if 'a' not in s
s
```

Out[5]:
```
{'b', 'c', 'd', 'r', 'z'}
```

In [6]:
```
s.remove('b')   # Raises KeyError if 'b' not in s
s
```

Out[6]:
```
{'c', 'd', 'r', 'z'}
```

Set operations

In [7]:
```
set('programming'), set('essentials')
```

Out[7]:
```
({'a', 'g', 'i', 'm', 'n', 'o', 'p', 'r'}, {'a', 'e', 'i', 'l', 'n', 's', 't'})
```

Union:

In [8]:
```
set('programming') | set('essentials')
```

Out[8]:
```
{'a', 'e', 'g', 'i', 'l', 'm', 'n', 'o', 'p', 'r', 's', 't'}
```

Intersection:

In [9]:
```
set('programming') & set('essentials')
```

Out[9]:
```
{'a', 'i', 'n'}
```

Difference:

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

Out[10]:
```
{'g', 'm', 'o', 'p', 'r'}
```

Symmetric difference:

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

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

Set "comparisons"

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

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

Out[12]:
```
False
```

Is one subset of another?

In [13]:
```
set('pro') <= set('programming')  # Or, set('pro').issubset(set('programming'))
```

Out[13]:
```
True
```

Is one superset of another?

In [14]:
```
set('pro') >= set('programming')  # Or, set('pro').issuperset(set('programming'))
```

Out[14]:
```
False
```

Set example: unique items

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

In [15]:
```
words = 'three one two one two one'.split()
print(words)
```

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

In [16]:
```
unique_words = list(set(words))
print(unique_words)
```

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

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

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 [17]:
```
orig_filenames = 'f1 f2 !prediction.txt f3 f4.ext !truth.txt f5'.split()
```

In [18]:
```
filenames = set(orig_filenames)
print(filenames)
for fname in {'!truth.txt', '!prediction.txt'}:
    filenames.discard(fname)
    print(filenames)
```

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

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 [19]:
```
filenames = set(orig_filenames)
print(filenames)
```

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

In [20]:
```
filenames = filenames - {'!truth.txt', '!prediction.txt'}
filenames
```

Out[20]:
```
{'f1', 'f2', 'f3', 'f4.ext', 'f5'}
```

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.

Dictionary

Creating a dictionary:

In [21]:
```
course = {'id': 'BE5B33PRG', 'name': 'Programming essentials', 'capacity': 25}
course2 = dict(id='BE5B33PRG', name='Programming essentials', capacity=25)
course3 = dict([('id', 'BE5B33PRG'), ('name', 'Programming essentials'), ('capacity', 2
5)])
course4 = dict(zip(('id', 'name', 'capacity'), ('BE5B33PRG', 'Programming essentials',
 25)))
```

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

In [22]:
```
course
```

Out[22]:
```
{'capacity': 25, 'id': 'BE5B33PRG', 'name': 'Programming essentials'}
```

In [23]:
```
course == course2 == course3 == course4
```

Out[23]:
```
True
```

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

In [24]:
```
'id' in course, 'BE5B33PRG' in course
```

Out[24]:
```
(True, False)
```

Querying a dictionary for a value:

In [25]:
```
course['id']
```

Out[25]:
```
'BE5B33PRG'
```

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

In [26]:
```
print(list(course.keys()))
print(list(course.values()))
print(list(course.items()))
```

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

Adding new key-value pairs:

In [27]:
```
course['lecturer'] = 'Svoboda'
print(course)
```

```
{'lecturer': 'Svoboda', 'name': 'Programming essentials', 'capacity': 25, 'id': 'BE5B
33PRG'}
```

Replacing a value for an existing key:

In [28]:
```
course['lecturer'] = 'Posik'
print(course)
```

```
{'lecturer': 'Posik', 'name': 'Programming essentials', 'capacity': 25, 'id': 'BE5B33
PRG'}
```

Removing an item from a dictionary:

In [29]:
```
del course['lecturer']
print(course)
```

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

Iterating over dictionaries

Iterating over keys:

In [30]:
```
for key in course:
    print(key, end=' | ')
```

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

or

In [31]:
```
for key in course.keys():
    print(key, end=' | ')
```

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

Iterating over values:

In [32]:
```
for val in course.values():
    print(val, end=' | ')
```

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

Iterating over key-value pairs:

In [33]:
```
for item in course.items():
    print(item[0], '=', item[1], end=' | ')
```

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

or, in a better way:

In [34]:
```
for key, val in course.items():
    print(key, '=', val, end=' | ')
```

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

`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 [35]:
```
print(course['id'])
```

```
BE5B33PRG
```

In [36]:
```
print(course.get('id'))
```

```
BE5B33PRG
```

Querying a value for a non-existent key:

In [37]:
```
course
```

Out[37]:
```
{'capacity': 25, 'id': 'BE5B33PRG', 'name': 'Programming essentials'}
```

In [38]:
```
#print(course['univ'])      # Raises KeyError
```

In [39]:
```
print(course.get('univ'))
```

```
None
```

In [40]:
```
print(course.get('univ', 'CTU in Prague'))
```

```
CTU in Prague
```

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
```

Creating a Counter

In [41]:
```
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
```

Accessing Counter elements

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

In [42]:
```
c = Counter(['eggs', 'ham'])
print(c)
```

```
Counter({'ham': 1, 'eggs': 1})
```

In [43]:
```
print(c['eggs'])
print(c['bacon'])
```

```
1
0
```

`Counter.most_common()`

In [44]:
```
c = Counter('abracadabra')
c
```

Out[44]:
```
Counter({'a': 5, 'b': 2, 'c': 1, 'd': 1, 'r': 2})
```

In [45]:
```
c.most_common(3)
```

Out[45]:
```
[('a', 5), ('b', 2), ('r', 2)]
```

Adding and subtracting counters

In [46]:
```
c1 = Counter('abracadabra')
c2 = Counter('simsalabim')
print(c1)
print(c2)
```

```
Counter({'a': 5, 'b': 2, 'r': 2, 'd': 1, 'c': 1})
Counter({'a': 2, 'm': 2, 'i': 2, 's': 2, 'l': 1, 'b': 1})
```

In [47]:
```
print(c1 + c2)
```

```
Counter({'a': 7, 'b': 3, 'i': 2, 'r': 2, 'm': 2, 's': 2, 'c': 1, 'd': 1, 'l': 1})
```

In [48]:
```
print(c1 - c2)
```

```
Counter({'a': 3, 'r': 2, 'c': 1, 'b': 1, 'd': 1})
```

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

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

In [49]:
```
c = Counter()
c1 = Counter('abrakadabra')
c2 = Counter('avada kedavra')
```

In [50]:
```
c.subtract(c1)     # Negative counts
print(c1)
print(c)
```

```
Counter({'a': 5, 'b': 2, 'r': 2, 'd': 1, 'k': 1})
Counter({'k': -1, 'd': -1, 'r': -2, 'b': -2, 'a': -5})
```

In [51]:
```
c.update(c2)
print(c)
```

```
Counter({'v': 2, ' ': 1, 'e': 1, 'd': 1, 'k': 0, 'a': 0, 'r': -1, 'b': -2})
```

In [52]:
```
c.update(c1)
c.subtract(c2)
print(c)
```

```
Counter({'v': 0, ' ': 0, 'r': 0, 'k': 0, 'e': 0, 'd': 0, 'b': 0, 'a': 0})
```

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.

Named tuple: Example 1

In [53]:
```
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)
```

```
2015-11-26 222 3
('2015-11-26', 222, 3)
2015-11-26 222 3
```

In [54]:
```
# 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))
```

```
Total: $30.00
```

Named tuple: Example 2

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

In [55]:
```
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)
```

```
Aircraft(manufacturer='Airbus', model='A320-200', seats=Seating(min=100, max=220))
220
```

Extracting items of named tuples for printing

In [56]:
```
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()))
```

```
Airbus A320-200
Airbus A320-200
Airbus A320-200
Airbus A320-200
Airbus A320-200
```

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!

Notebook config

Some setup follows. Ignore it.

In [57]:
```
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
})
```

Out[57]:
```
{'height': 768,
 'minScale': 1.0,
 'start_slideshow_at': 'selected',
 'theme': 'Simple',
 'transition': 'slide',
 'width': 1268}
```

In [58]:
```
%%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, monospace;
}
pre, code, .CodeMirror-code {
    font-size: inherit !important;
}
.reveal .code_cell {
    font-size: 130% !important;
    line-height: 130% !important;
}
</style>
```