Lecture 6 – Collections, sets, dictionaries

https://cw.fel.cvut.cz/wiki/courses/be5b33prg/start

Michal Reinštein

Czech Technical University in Prague,
Faculty of Electrical Engineering, Dept. of Cybernetics,
Center for Machine Perception
http://cmp.felk.cvut.cz/~reinsmic/
reinstein.michal@fel.cvut.cz

ADMIN

- No classes on 17.11. 2017 due to public holiday
- Prepare for the **mid-term test** 24.11. 2017 during labs
- Preliminary exam dates (check the faculty system):

  19.1. 2018

  26.1. 2018

  2.2. 2018

  9.2. 2018

  Multiple choice test, **no materials** or **devices** allowed.

  The exam starts sharp at 8:00.

  Use the Faculty information system to enroll (find the room!).

source courtesy of Petr Posik BE5b33PR 2016/2017

SEQUENCE TYPES

- Sequences of items– they support
  - membership operator **in**
  - querying for size **len**
  - indexing and slices **[]**
  - Iterable

- **string**: immutable ordered sequence of characters
- **tuple**: immutable ordered sequence of items of any data type
- **list**: mutable ordered sequence of items of any data type

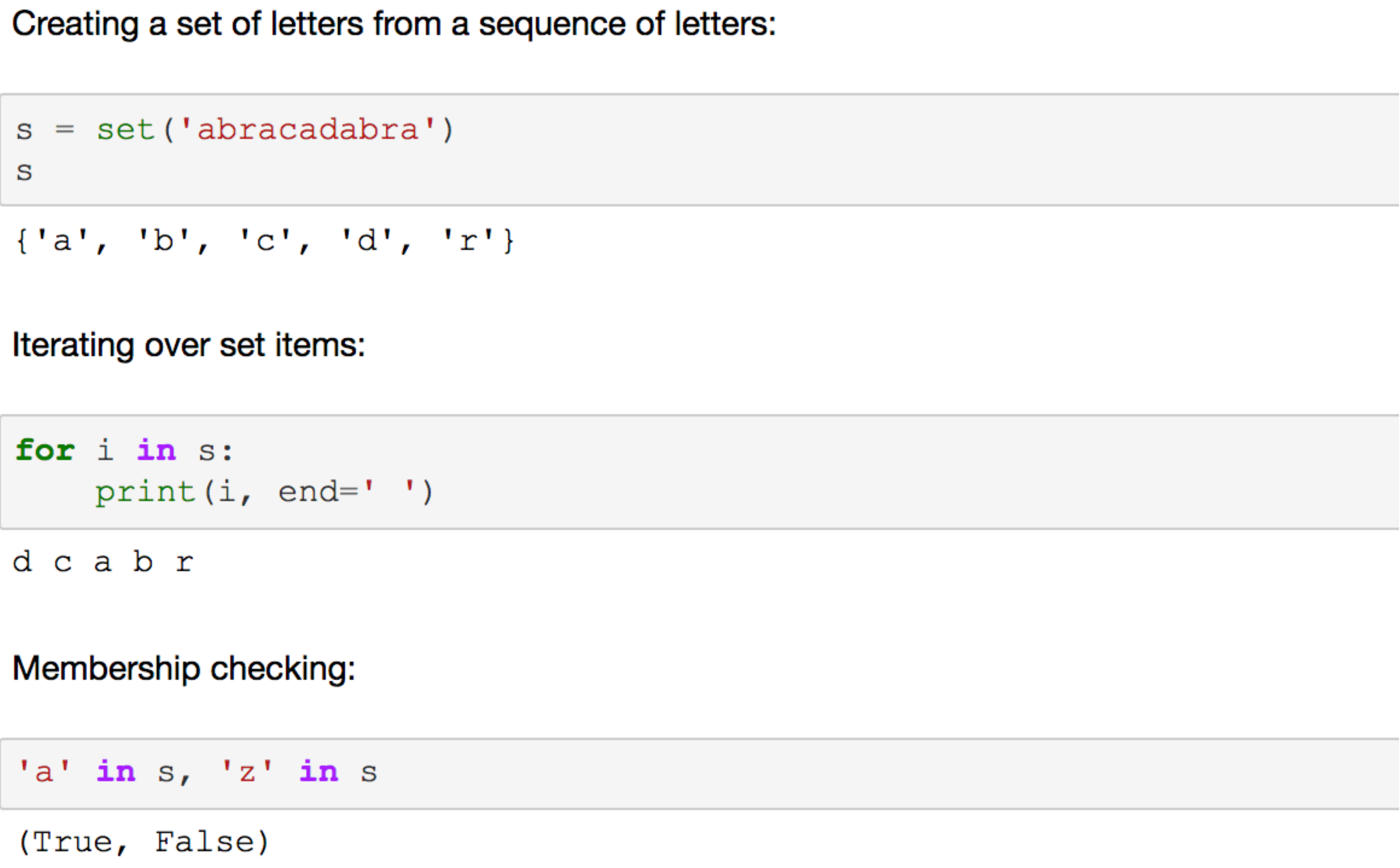
source courtesy of Petr Posik BE5b33PR 2016/2017

SET TYPES

- Set types support
  - membership operator **in**
  - querying for size **len**
  - iterable
  - set operations (*comparisons, union, intersection, subset*)
- **Set**: mutable unordered collection of unique items of any type
- **Frozenset**: immutable unordered collection of unique items of any data type
- When iterated over, sets provide items in an arbitrary order
- Only **hashable** objects may be added to a set:
  - Immutable data types are hashable (hash value dos not change, compare for equality to other objects) (*int, float, str, tuple, frozenset*)
  - Mutable values are (usually) not hashable (*list, dict, set*)

source courtesy of Petr Posik BE5b33PR 2016/2017

SET USAGE



Creating a set of letters from a sequence of letters:

```
s = set('abracadabra')
s
```

```
{'a', 'b', 'c', 'd', 'r'}
```

Iterating over set items:

```
for i in s:
    print(i, end=' ')
```

```
d c a b r
```

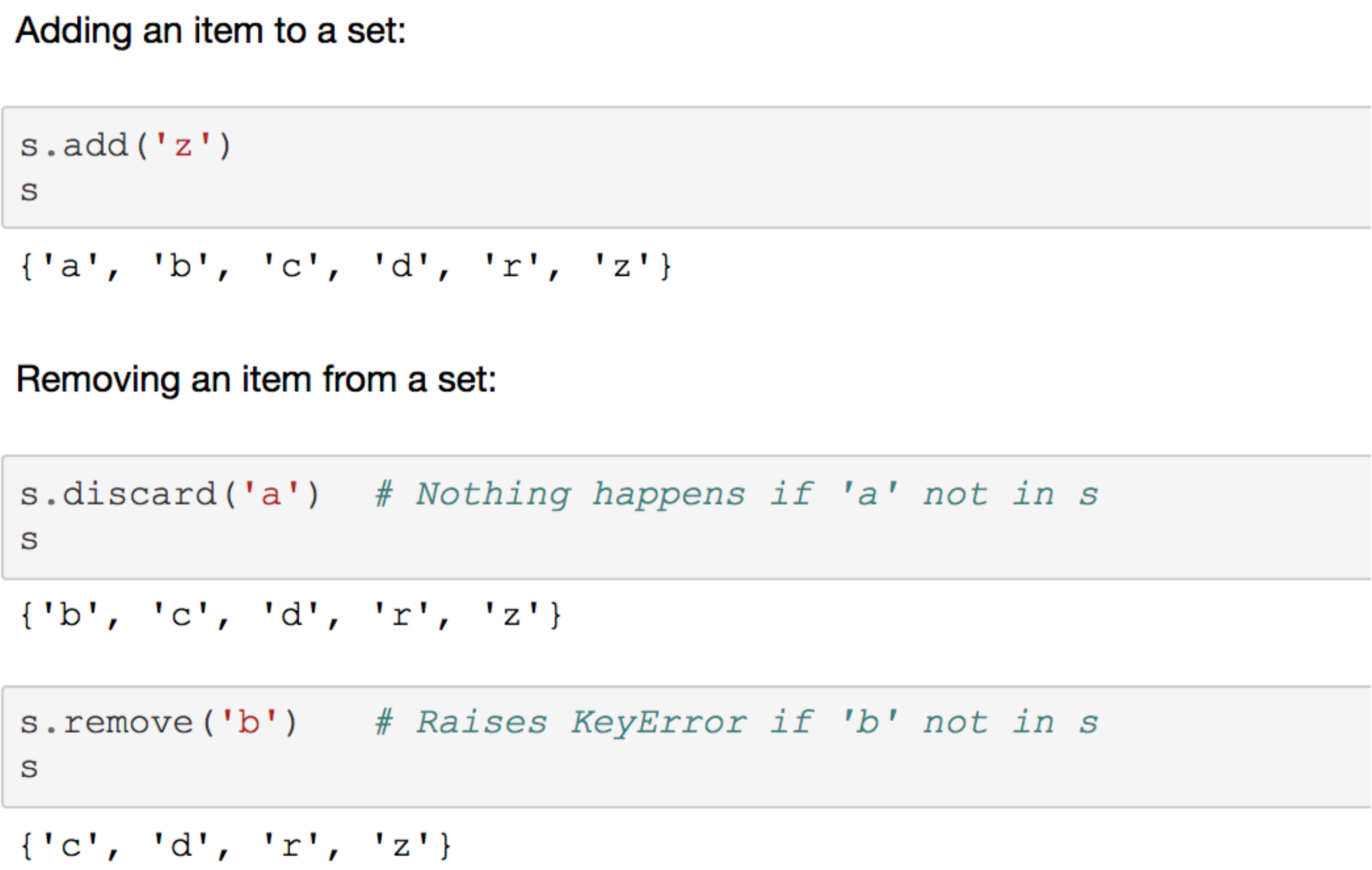
Membership checking:

```
'a' in s, 'z' in s
```

```
(True, False)
```

source courtesy of Petr Posik BE5b33PR 2016/2017

SET USAGE



Adding an item to a set:

```
s.add('z')
s
```

```
{'a', 'b', 'c', 'd', 'r', 'z'}
```

Removing an item from a set:

```
s.discard('a')   # Nothing happens if 'a' not in s
s
```

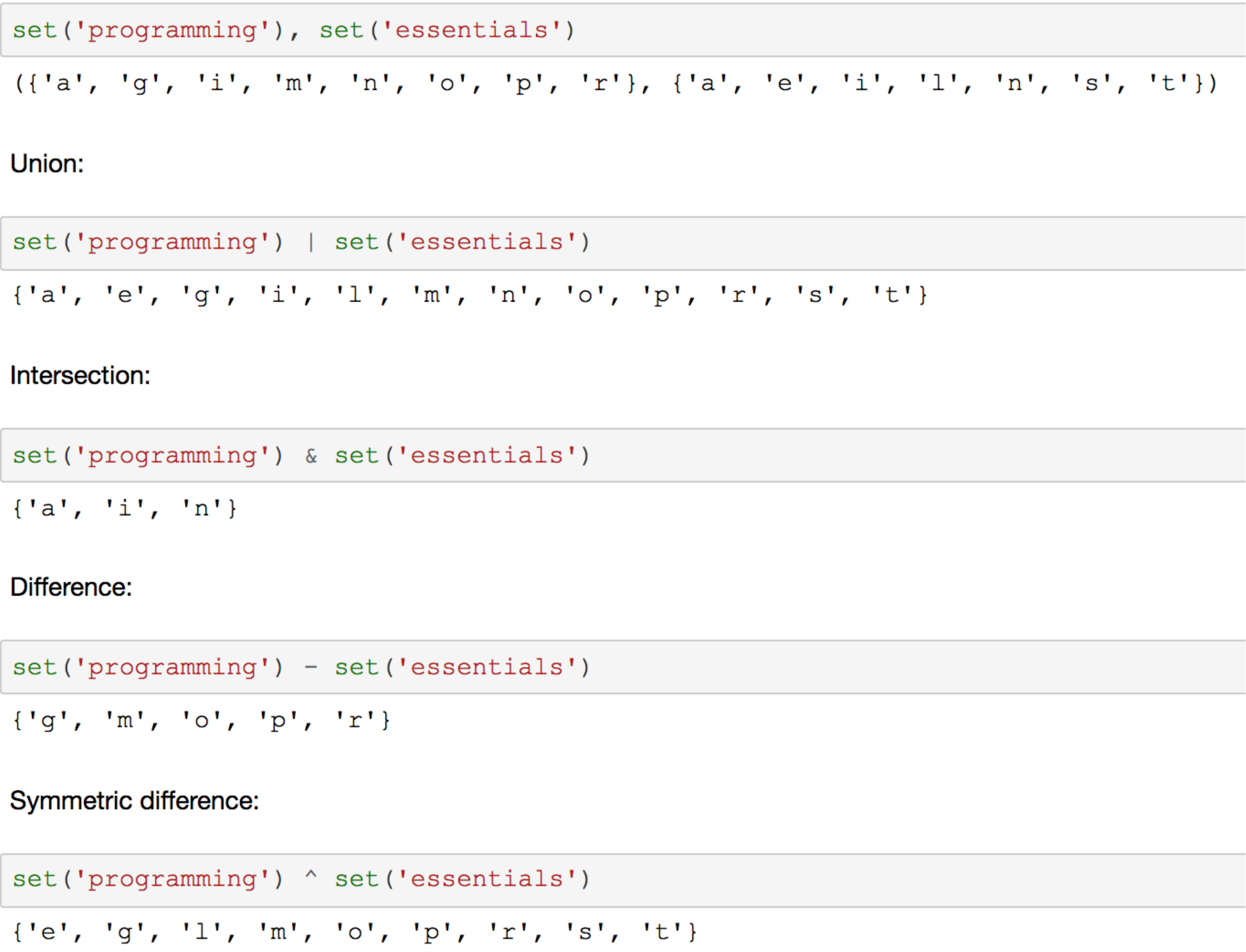
```
{'b', 'c', 'd', 'r', 'z'}
```

```
s.remove('b')    # Raises KeyError if 'b' not in s
s
```

```
{'c', 'd', 'r', 'z'}
```

source courtesy of Petr Posik BE5b33PR 2016/2017

SET OPERATIONS



```
set('programming'), set('essentials')
```

```
({'a', 'g', 'i', 'm', 'n', 'o', 'p', 'r'}, {'a', 'e', 'i', 'l', 'n', 's', 't'})
```

Union:

```
set('programming') | set('essentials')
```

```
{'a', 'e', 'g', 'i', 'l', 'm', 'n', 'o', 'p', 'r', 's', 't'}
```

Intersection:

```
set('programming') & set('essentials')
```

```
{'a', 'i', 'n'}
```

Difference:

```
set('programming') - set('essentials')
```

```
{'g', 'm', 'o', 'p', 'r'}
```

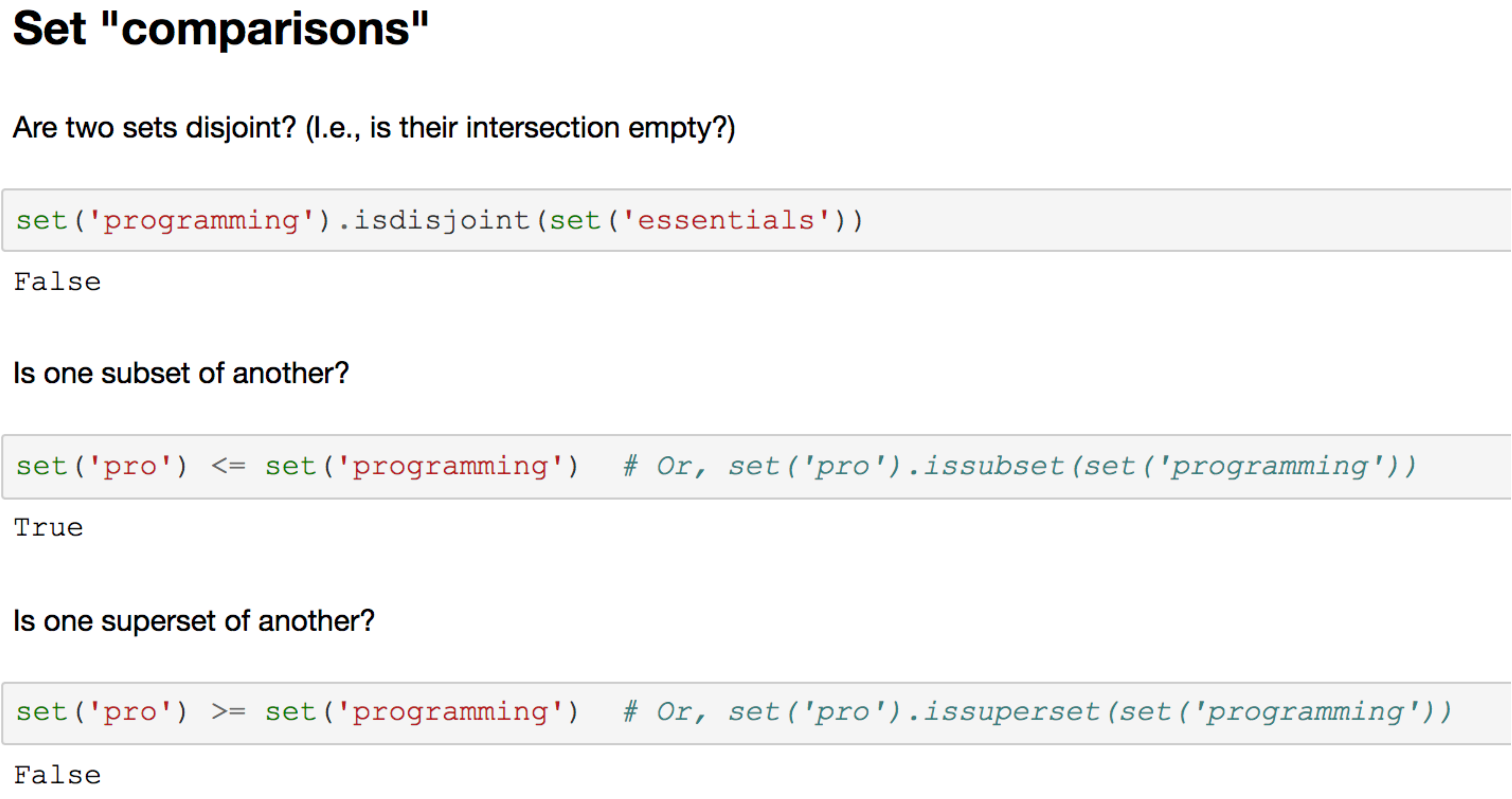
Symmetric difference:

```
set('programming') ^ set('essentials')
```

```
{'e', 'g', 'l', 'm', 'o', 'p', 'r', 's', 't'}
```

source courtesy of Petr Posik BE5b33PR 2016/2017

SET OPERATIONS



Set "comparisons"

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

```
set('programming').isdisjoint(set('essentials'))
```

```
False
```

Is one subset of another?

```
set('pro') <= set('programming')   # Or, set('pro').issubset(set('programming'))
```

```
True
```

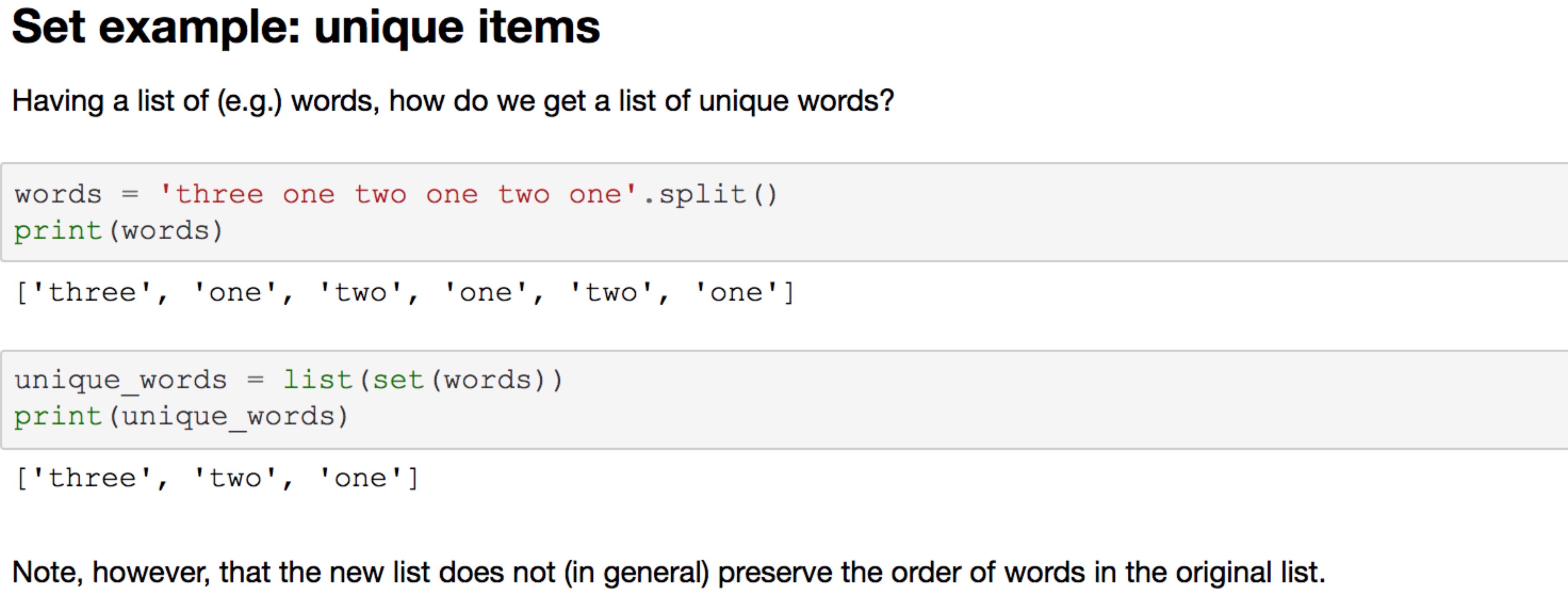
Is one superset of another?

```
set('pro') >= set('programming')   # Or, set('pro').issuperset(set('programming'))
```

```
False
```

source courtesy of Petr Posik BE5b33PR 2016/2017

SET OPERATIONS



Set example: unique items

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

```
words = 'three one two one two one'.split()
print(words)
```

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

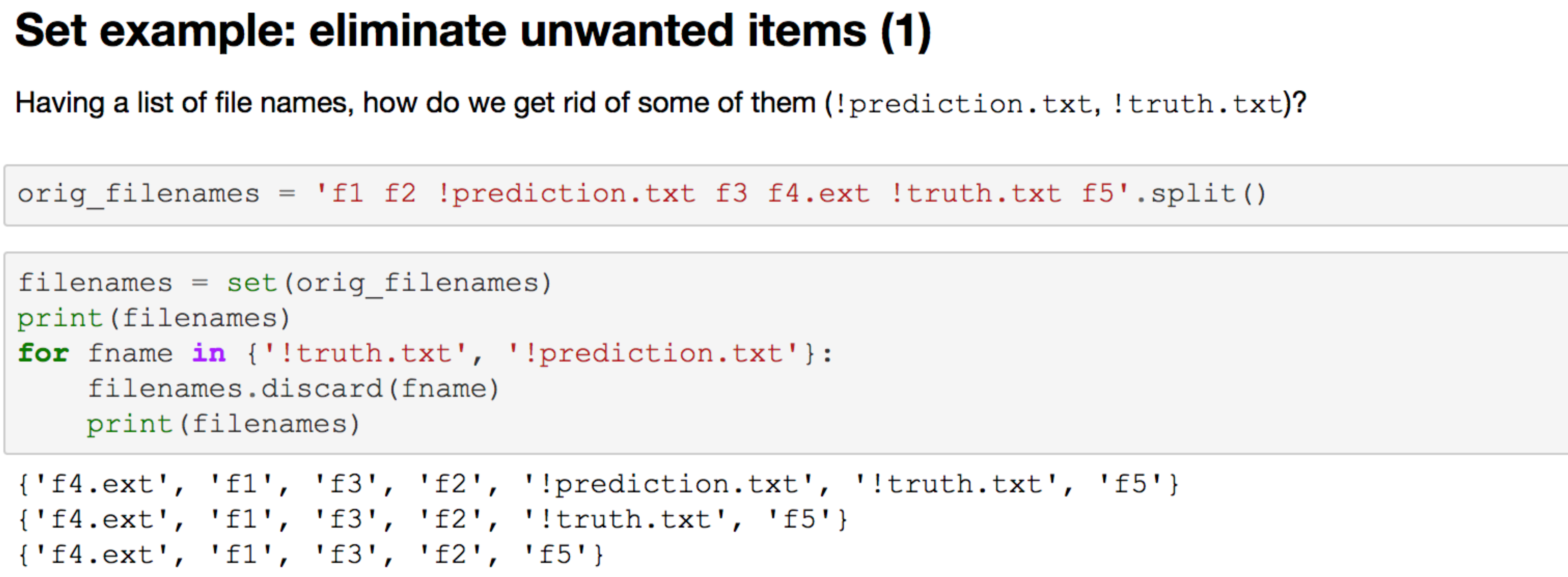
```
unique_words = list(set(words))
print(unique_words)
```

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

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

source courtesy of Petr Posik BE5b33PR 2016/2017

SET OPERATIONS



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`)?

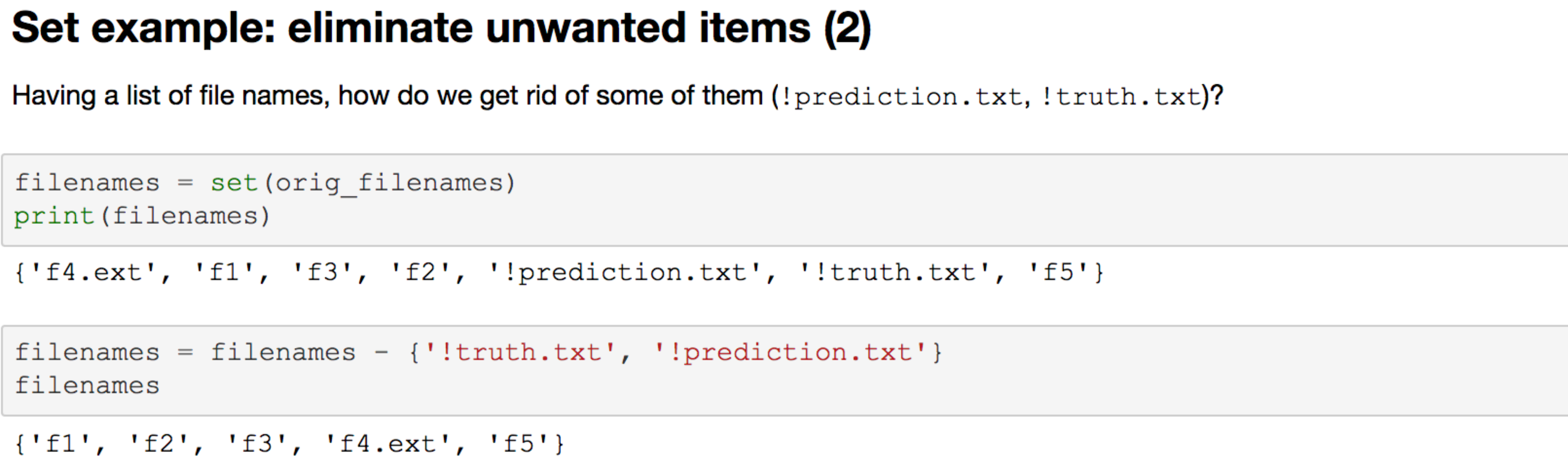
```
orig_filenames = 'f1 f2 !prediction.txt f3 f4.ext !truth.txt f5'.split()
```

```
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'}
```

source courtesy of Petr Posik BE5b33PR 2016/2017

SET OPERATIONS



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`)?

```
filenames = set(orig_filenames)
print(filenames)
```

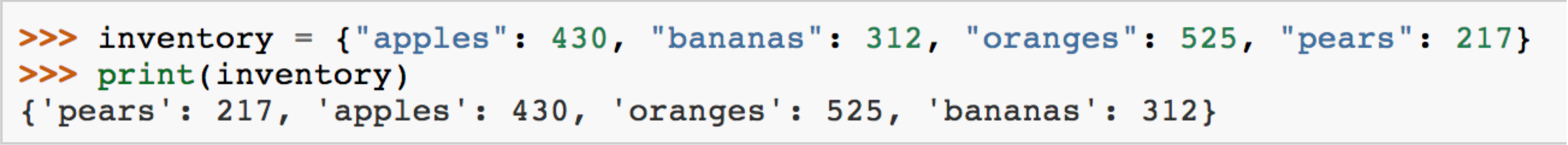
```
{'f4.ext', 'f1', 'f3', 'f2', '!prediction.txt', '!truth.txt', 'f5'}
```

```
filenames = filenames - {'!truth.txt', '!prediction.txt'}
filenames
```

```
{'f1', 'f2', 'f3', 'f4.ext', 'f5'}
```

source courtesy of Petr Posik BE5b33PR 2016/2017

MAPPING TYPES

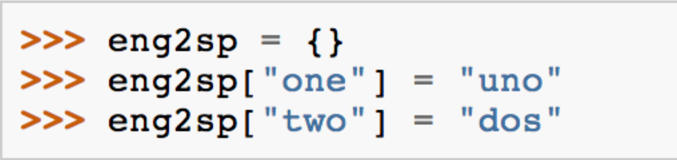


```
>>> inventory = {"apples": 430, "bananas": 312, "oranges": 525, "pears": 217}
>>> print(inventory)
{'pears': 217, 'apples': 430, 'oranges': 525, 'bananas': 312}
```

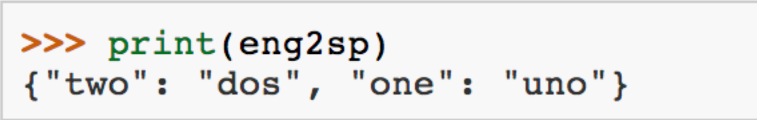
- A mapping type is an **unordered collection of key-value pairs**
- They support
  - membership operator **in**
  - querying for size **len**
  - iterable
- Only **hashable** (i.e. immutable) objects can be used as keys
- Each key's associated value may be of **any data type**

source courtesy of Petr Posik BE5b33PR 2016/2017

DICTIONARIES



```
>>> eng2sp = {}
>>> eng2sp["one"] = "uno"
>>> eng2sp["two"] = "dos"
```



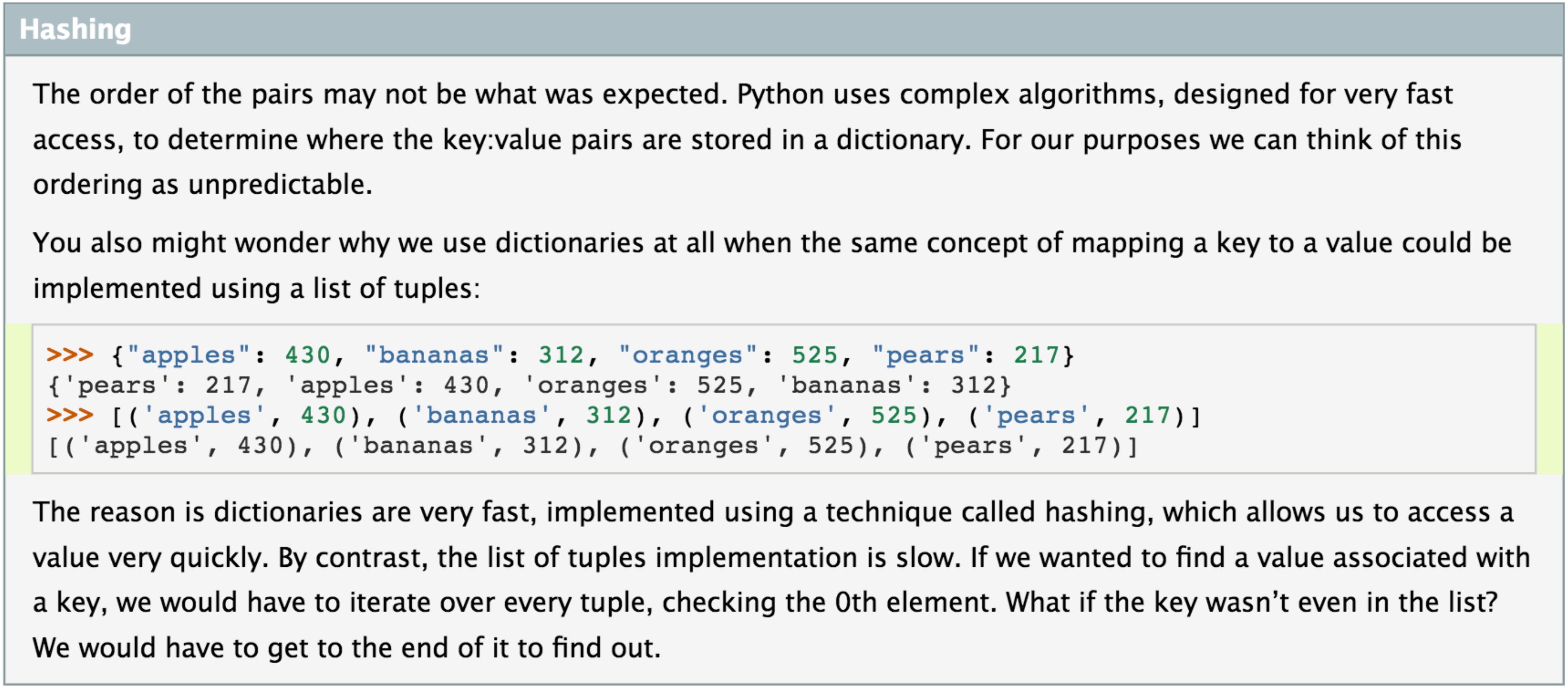
```
>>> print(eng2sp)
{"two": "dos", "one": "uno"}
```

- Strings, lists, and tuples — are sequence types *using integers as indices* to access the values they contain within them
- Dictionaries are Python's built-in **mapping type**
- They map **keys** (**any immutable type**) to **values** that can be any type (**heterogeneous**)
- Other languages: associative arrays (associate key with value)

- EXAMPLE: *Create a dictionary to translate English words into Spanish (the keys are strings). One way to create a dictionary is to start with the empty dictionary and add key : value pairs.*
- The **empty dictionary** is denoted **{}**

source courtesy of Petr Posik BE5b33PR 2016/2017

DICTIONARIES



Hashing

The order of the pairs may not be what was expected. Python uses complex algorithms, designed for very fast access, to determine where the key:value pairs are stored in a dictionary. For our purposes we can think of this ordering as unpredictable.

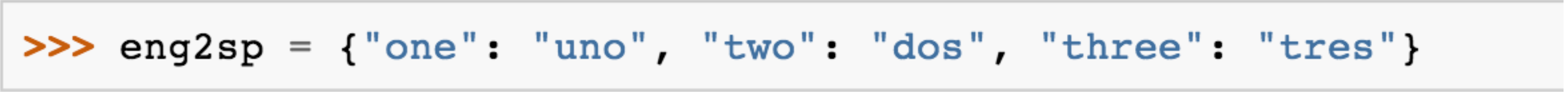
You also might wonder why we use dictionaries at all when the same concept of mapping a key to a value could be implemented using a list of tuples:

```
>>> {"apples": 430, "bananas": 312, "oranges": 525, "pears": 217}
{'pears': 217, 'apples': 430, 'oranges': 525, 'bananas': 312}
>>> [('apples', 430), ('bananas', 312), ('oranges', 525), ('pears', 217)]
[('apples', 430), ('bananas', 312), ('oranges', 525), ('pears', 217)]
```

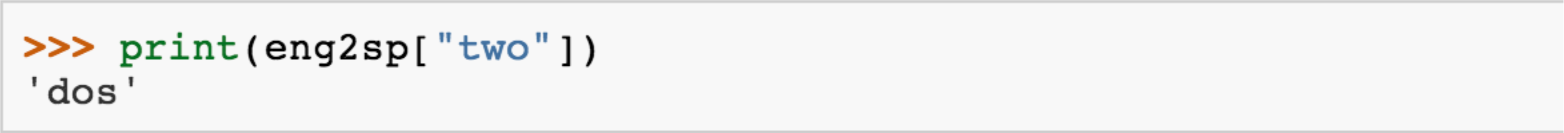
The reason is dictionaries are very fast, implemented using a technique called hashing, which allows us to access a value very quickly. By contrast, the list of tuples implementation is slow. If we wanted to find a value associated with a key, we would have to iterate over every tuple, checking the 0th element. What if the key wasn’t even in the list? We would have to get to the end of it to find out.

source http://openbookproject.net/thinkcs/python/english3e/dictionaries.html

DICTIONARIES



```
>>> eng2sp = {"one": "uno", "two": "dos", "three": "tres"}
```

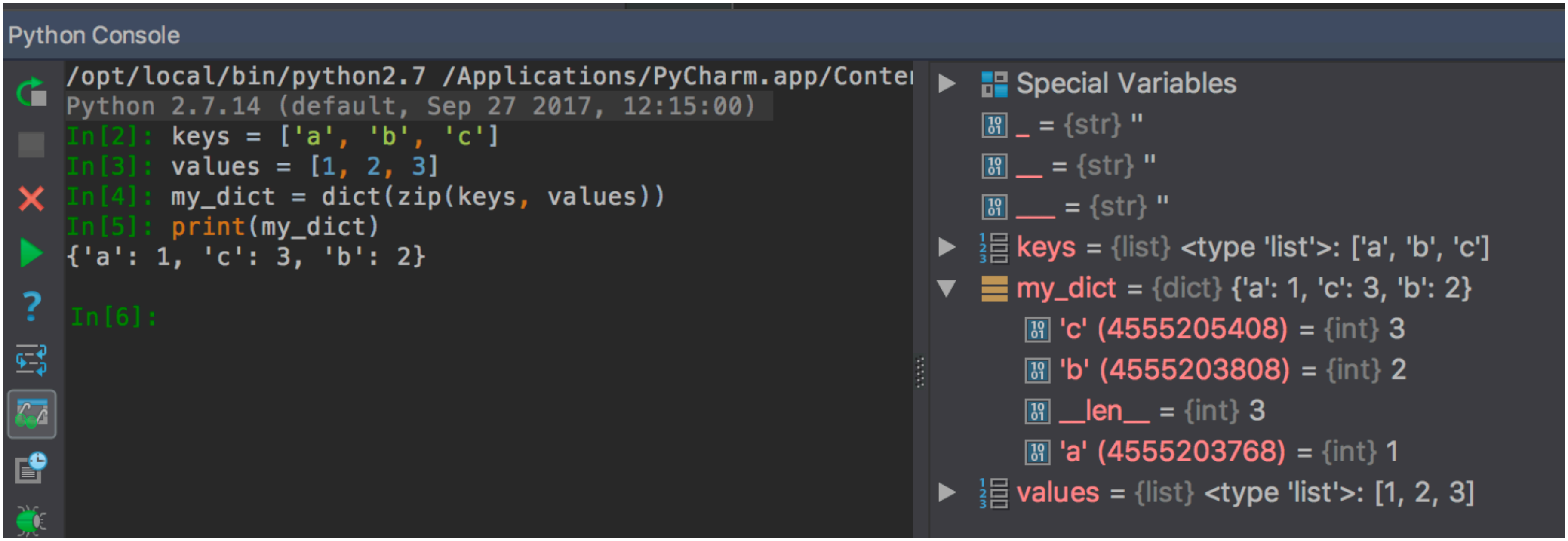


```
>>> print(eng2sp["two"])
'dos'
```

- To create a dictionary is to provide a list of **key:value pairs** using the same syntax as the previous output
- Order of pairs does not matter – the values in a dictionary are accessed with keys, not with indices, **no order guaranteed**
- Key is used to look up the corresponding value: *the key "two" yields the value "dos"*
- *Lists*, *tuples*, and *strings* have been called *sequences*, because their items occur in order
- The dictionary is compound type that is **not a sequence** (**no indexing or slicing**)

source http://openbookproject.net/thinkcs/python/english3e/dictionaries.html

DICTIONARIES



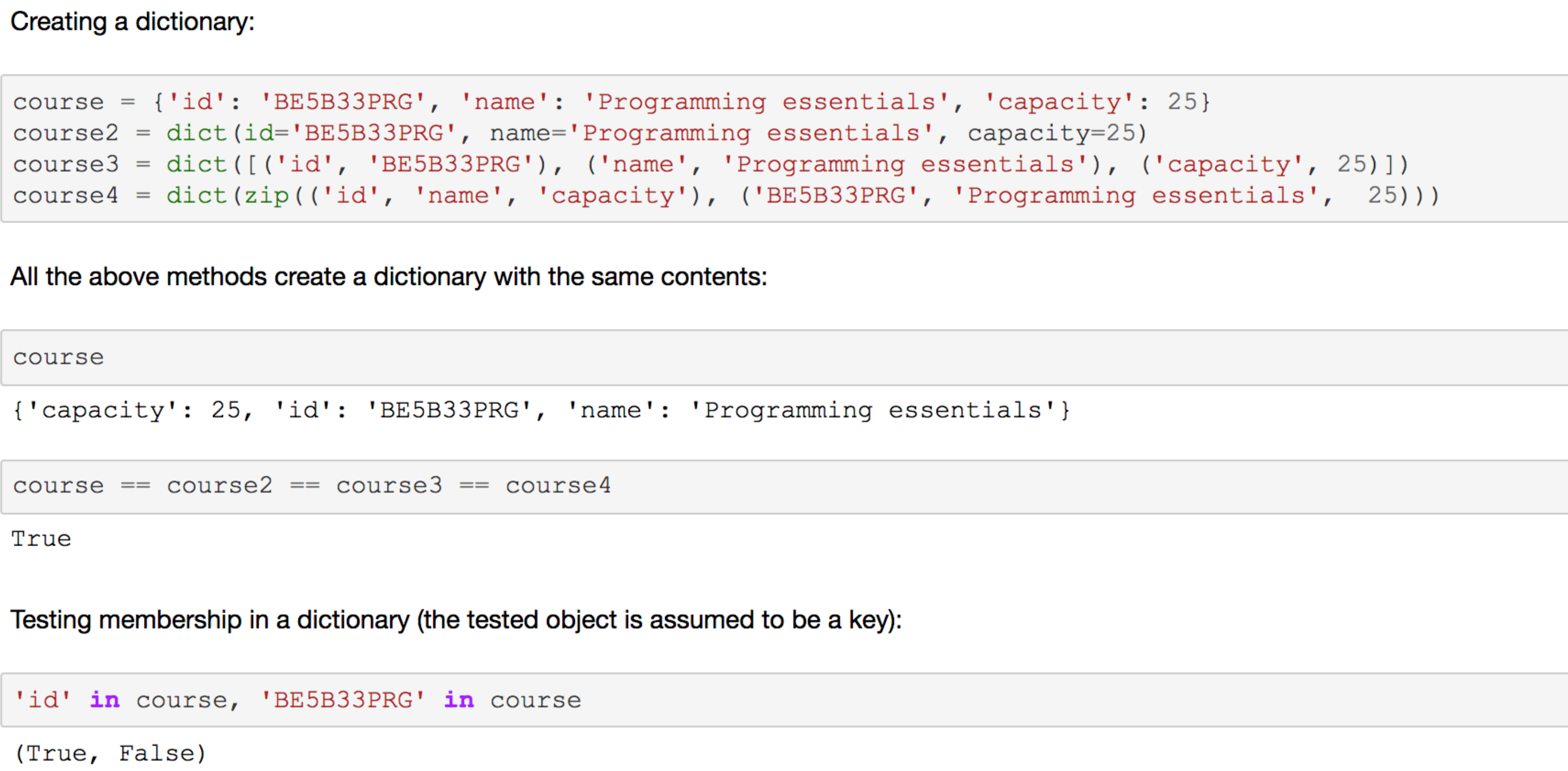
```
/opt/local/bin/python2.7 /Applications/PyCharm.app/Conte
Python 2.7.14 (default, Sep 27 2017, 12:15:00)
In[2]: keys = ['a', 'b', 'c']
In[3]: values = [1, 2, 3]
In[4]: my_dict = dict(zip(keys, values))
In[5]: print(my_dict)
{'a': 1, 'c': 3, 'b': 2}

In[6]:
```

```
▶ Special Variables
    _ = {str} ''
    __ = {str} ''
    ___ = {str} ''
▶ keys = {list} <type 'list'>: ['a', 'b', 'c']
▼ my_dict = {dict} {'a': 1, 'c': 3, 'b': 2}
      'c' (4555205408) = {int} 3
      'b' (4555203808) = {int} 2
      __len__ = {int} 3
      'a' (4555203768) = {int} 1
▶ values = {list} <type 'list'>: [1, 2, 3]
```

- Keys and values can be defined as separate lists (*order matters!*)
- Lists can be paired using **zip**
- Once paired a dictionary can be created using **dict**

source http://openbookproject.net/thinkcs/python/english3e/dictionaries.html

DICTIONARIES



Creating a dictionary:

```
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', 25)])
course4 = dict(zip(('id', 'name', 'capacity'), ('BE5B33PRG', 'Programming essentials',  25)))
```

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

```
course
```

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

```
course == course2 == course3 == course4
```

```
True
```

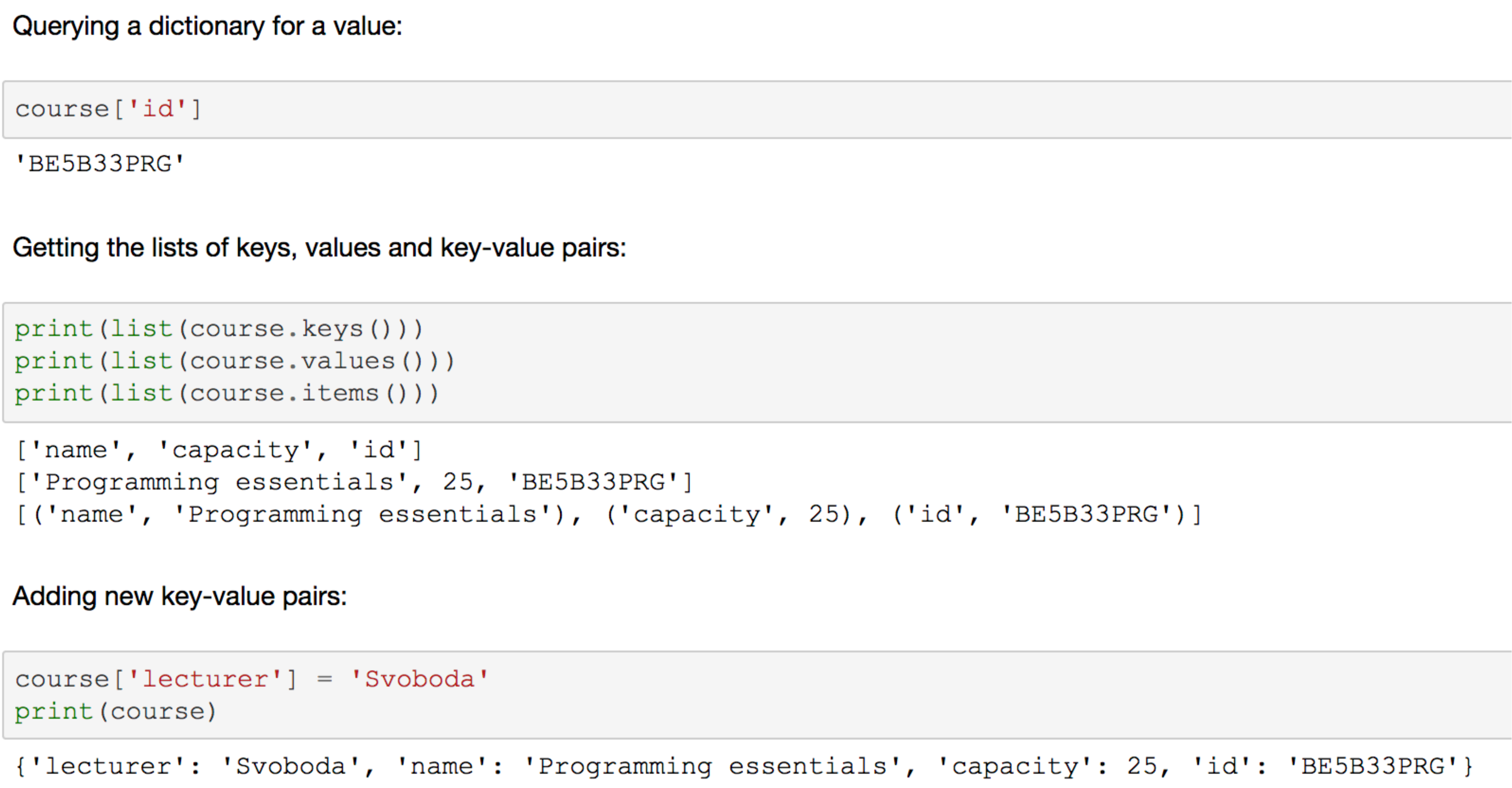
Testing membership in a dictionary (the tested object is assumed to be a key):

```
'id' in course, 'BE5B33PRG' in course
```

```
(True, False)
```

source courtesy of Petr Posik PRG 2016/2017

DICTIONARIES



Querying a dictionary for a value:

```
course['id']
```

```
'BE5B33PRG'
```

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

```
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')]
```

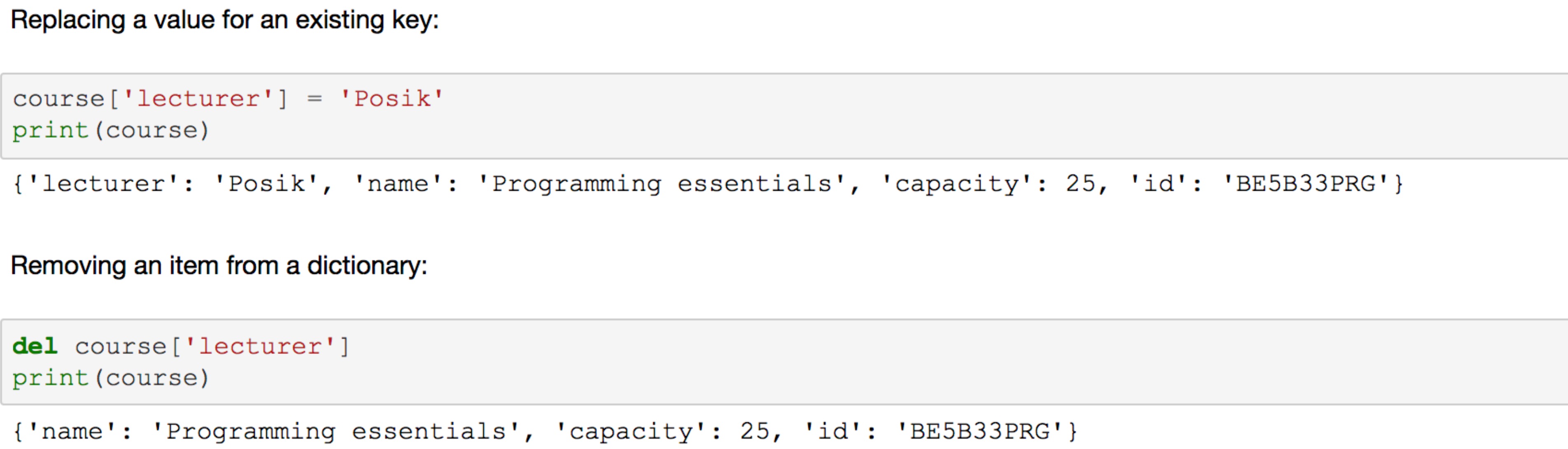
Adding new key-value pairs:

```
course['lecturer'] = 'Svoboda'
print(course)
```

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

source courtesy of Petr Posik PRG 2016/2017

DICTIONARIES



Replacing a value for an existing key:

```
course['lecturer'] = 'Posik'
print(course)
```

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

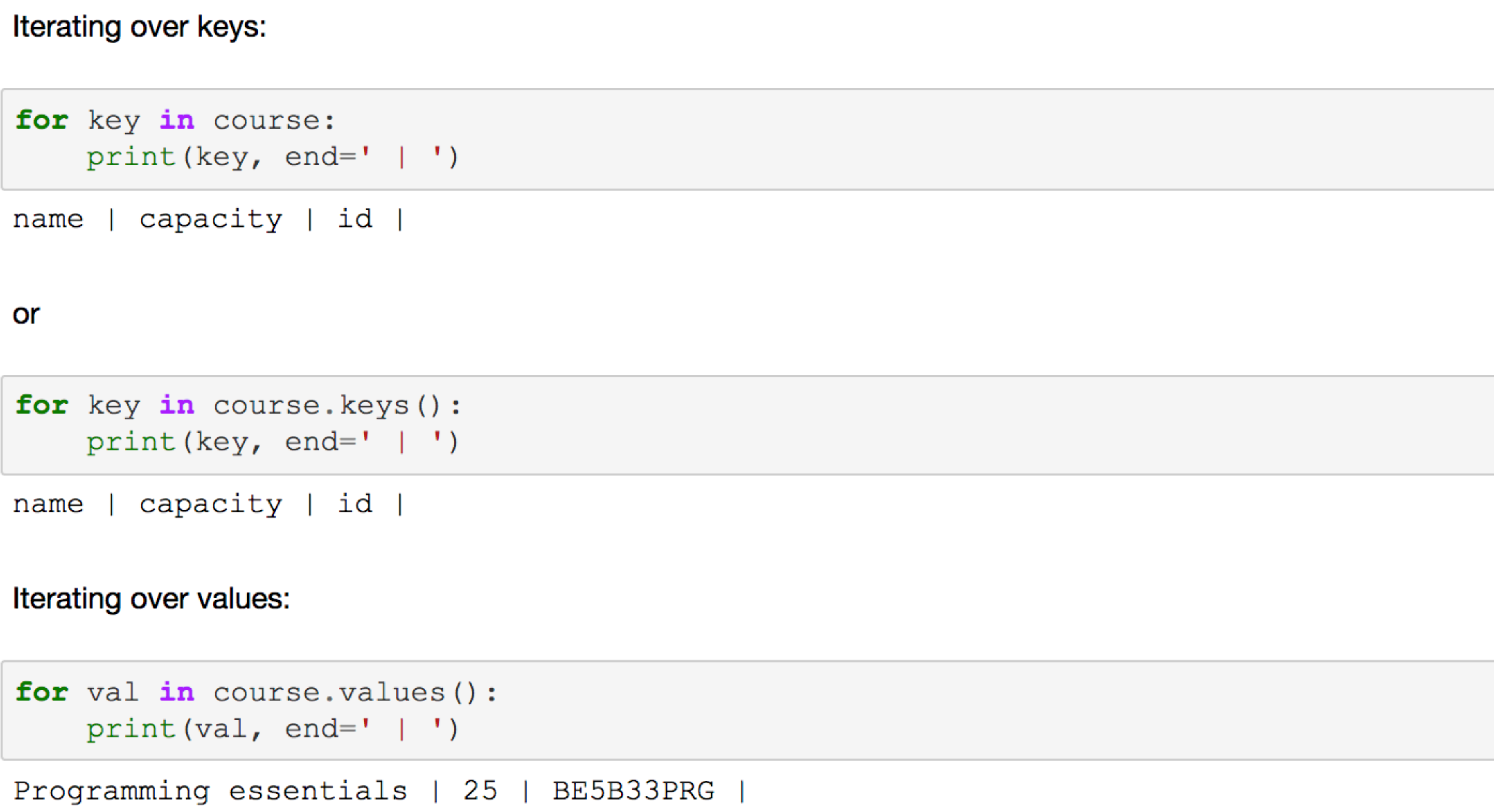
Removing an item from a dictionary:

```
del course['lecturer']
print(course)
```

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

source courtesy of Petr Posik PRG 2016/2017

DICTIONARIES



Iterating over keys:

```
for key in course:
    print(key, end=' | ')
```

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

or

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

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

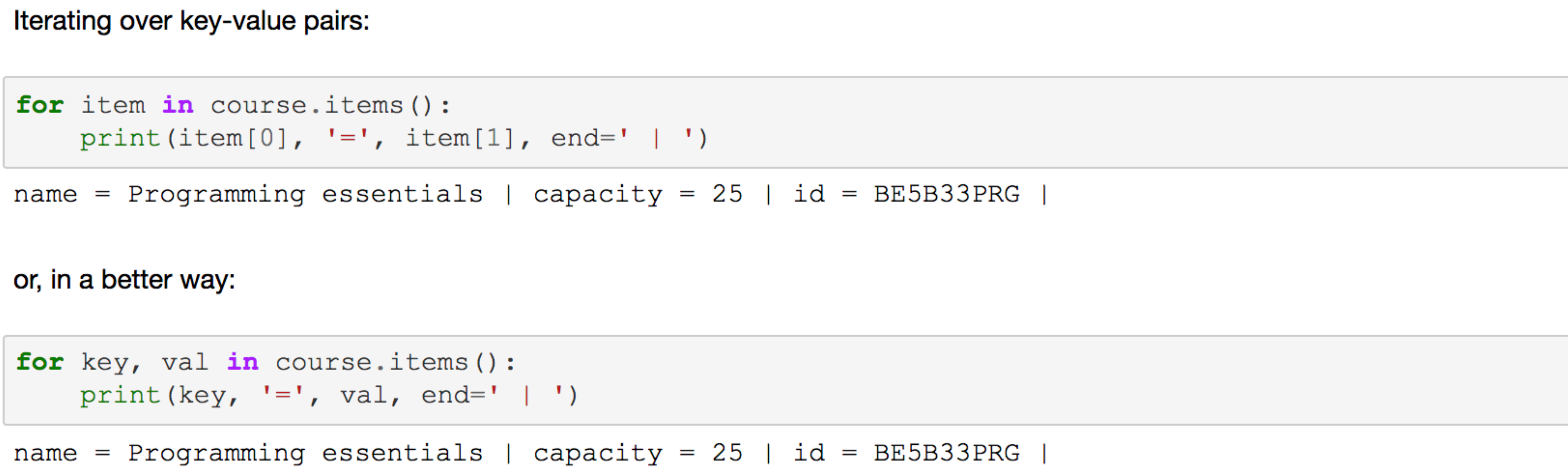
Iterating over values:

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

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

source courtesy of Petr Posik PRG 2016/2017

DICTIONARIES



Iterating over key-value pairs:

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

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

or, in a better way:

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

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

source courtesy of Petr Posik PRG 2016/2017

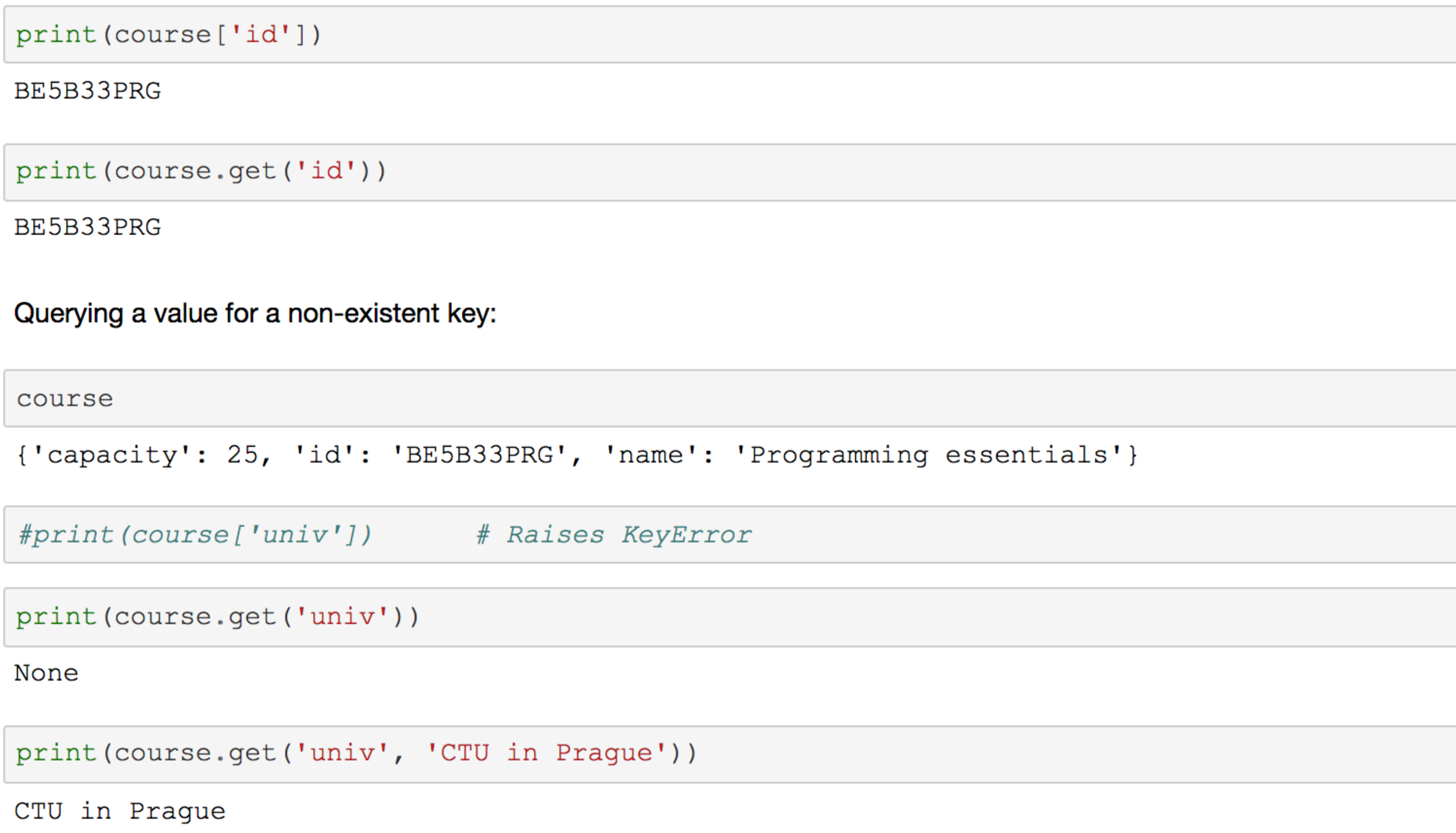
DICTIONARIES

dict.get() method

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

source courtesy of Petr Posik PRG 2016/2017

DICTIONARIES



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

```
BE5B33PRG
```

```
print(course.get('id'))
```

```
BE5B33PRG
```

Querying a value for a non-existent key:

```
course
```

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

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

```
print(course.get('univ'))
```

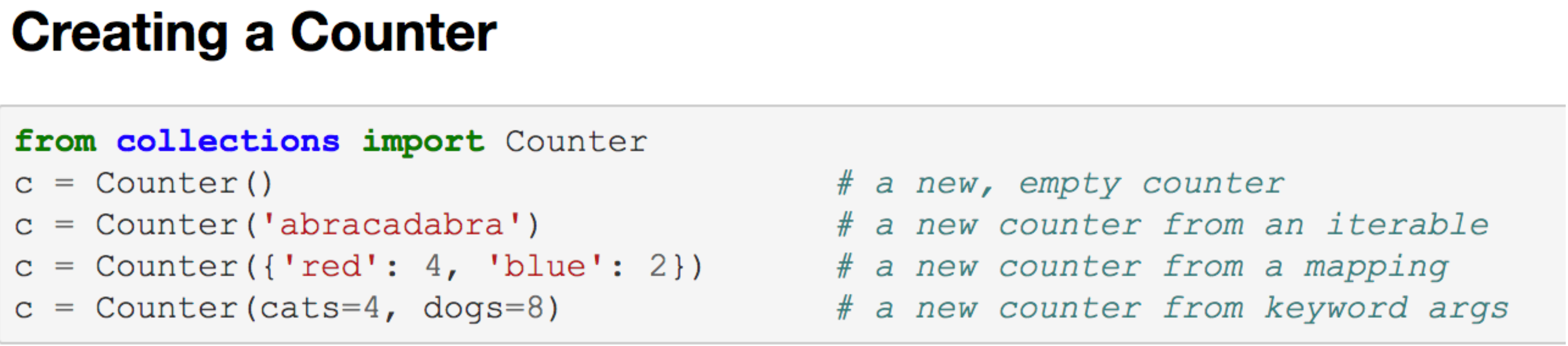
```
None
```

```
print(course.get('univ', 'CTU in Prague'))
```

```
CTU in Prague
```

source courtesy of Petr Posik PRG 2016/2017

DICTIONARIES



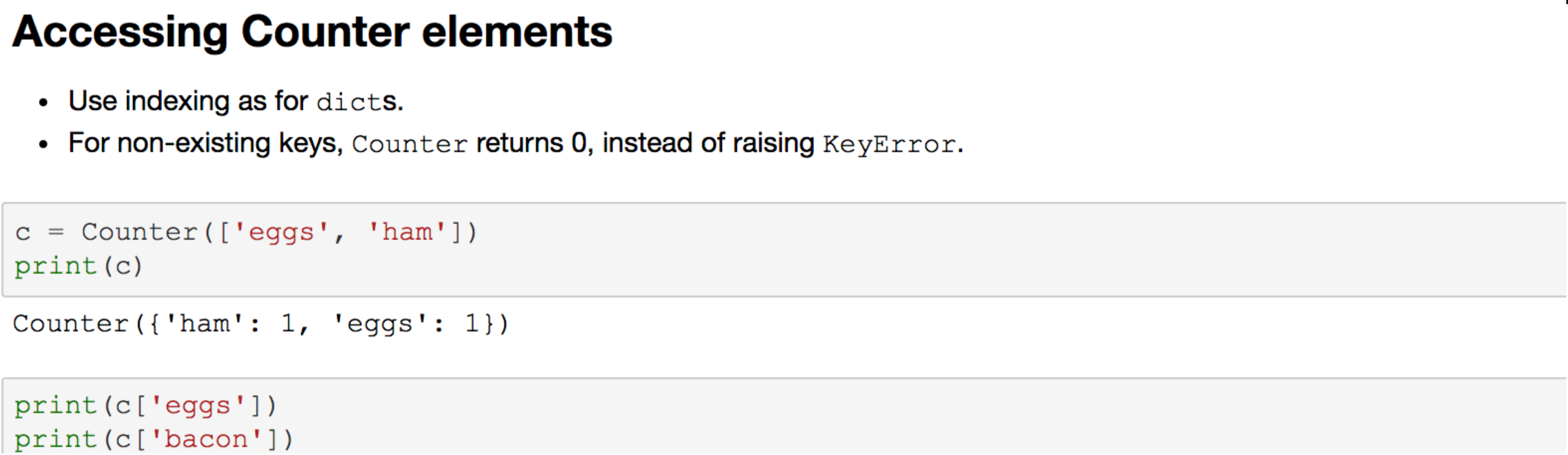
Creating a Counter

```
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
```

- Counter is 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**

source courtesy of Petr Posik PRG 2016/2017

DICTIONARIES



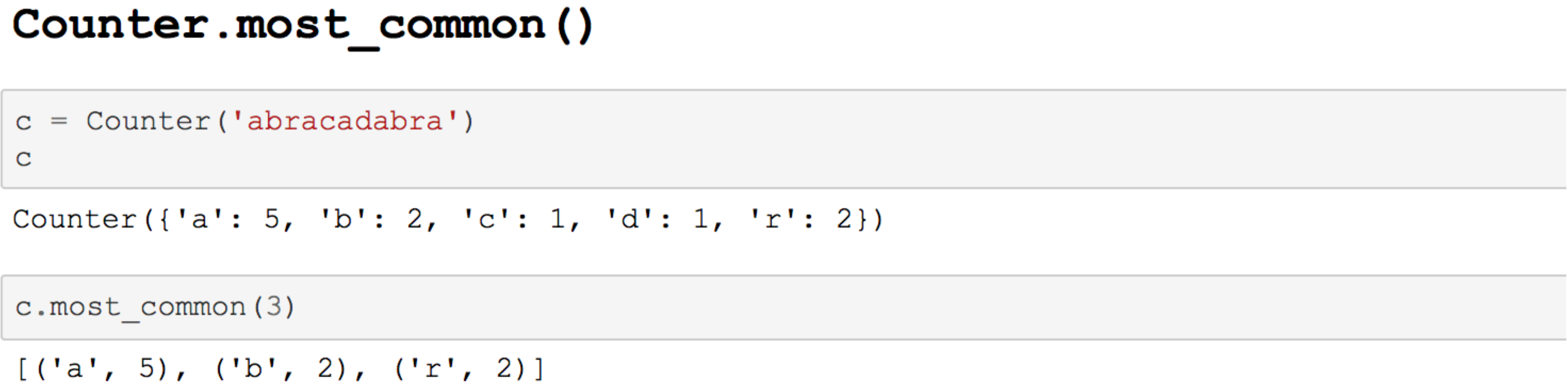
Accessing Counter elements

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

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

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

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



`Counter.most_common()`

```
c = Counter('abracadabra')
c
```

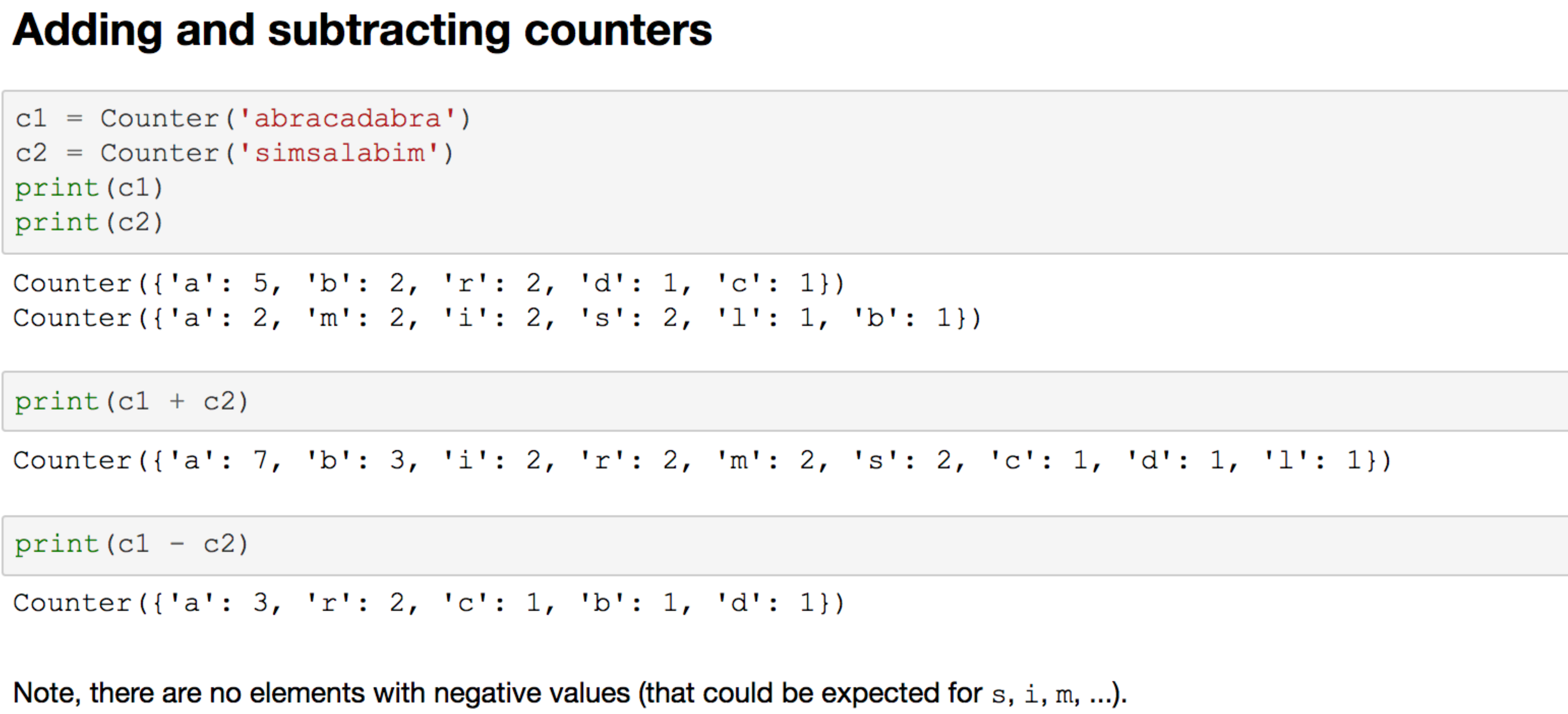
```
Counter({'a': 5, 'b': 2, 'c': 1, 'd': 1, 'r': 2})
```

```
c.most_common(3)
```

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

source courtesy of Petr Posik PRG 2016/2017

DICTIONARIES



Adding and subtracting counters

```
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})
```

```
print(c1 + c2)
```

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

```
print(c1 - c2)
```

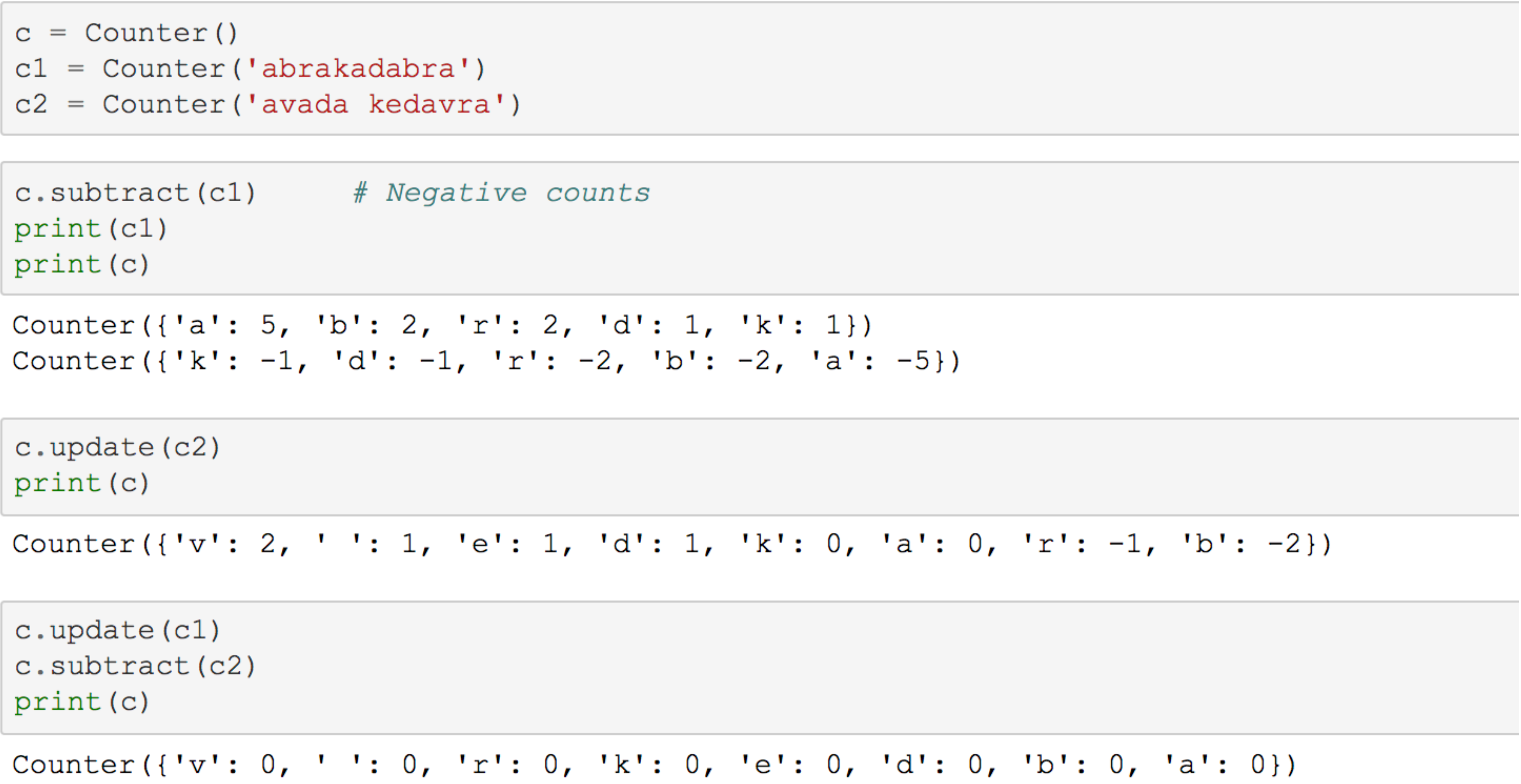
```
Counter({'a': 3, 'r': 2, 'c': 1, 'b': 1, 'd': 1})
```

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

source courtesy of Petr Posik PRG 2016/2017

DICTIONARIES

Counter.update() and Counter.subtract()



```
c = Counter()
c1 = Counter('abrakadabra')
c2 = Counter('avada kedavra')
```

```
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})
```

```
c.update(c2)
print(c)
```

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

```
c.update(c1)
c.subtract(c2)
print(c)
```

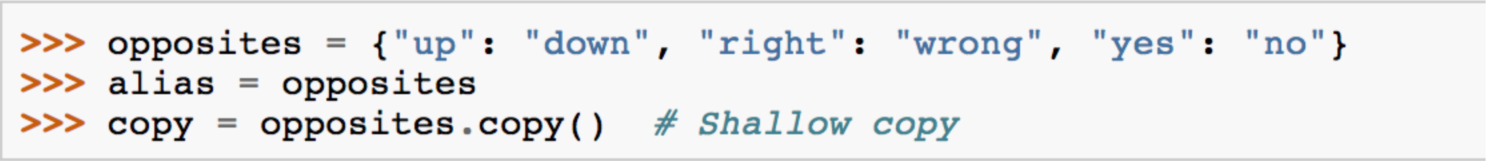
```
Counter({'v': 0, ' ': 0, 'r': 0, 'k': 0, 'e': 0, 'd': 0, 'b': 0, 'a': 0})
```

source courtesy of Petr Posik PRG 2016/2017

DICTIONARIES

- As in the case of lists, because **dictionaries are mutable**, we need to be aware of **aliasing**
- *Whenever two variables refer to the same object, changes to one affect the other*
- If we want to modify a dictionary and keep a copy of the original, **use the copy method**

- EXAMPLE: opposites is a dictionary that contains pairs of opposites

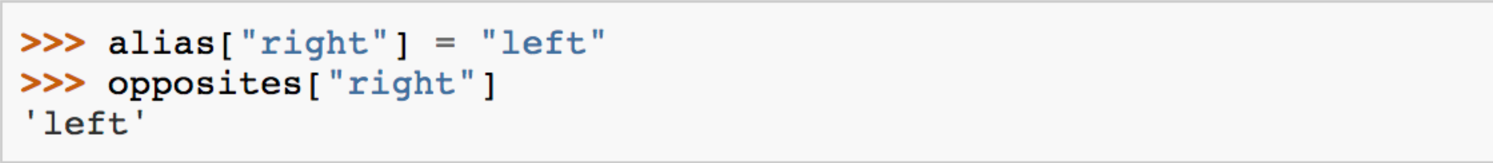


```
>>> opposites = {"up": "down", "right": "wrong", "yes": "no"}
>>> alias = opposites
>>> copy = opposites.copy()  # Shallow copy
```

source http://openbookproject.net/thinkcs/python/english3e/dictionaries.html

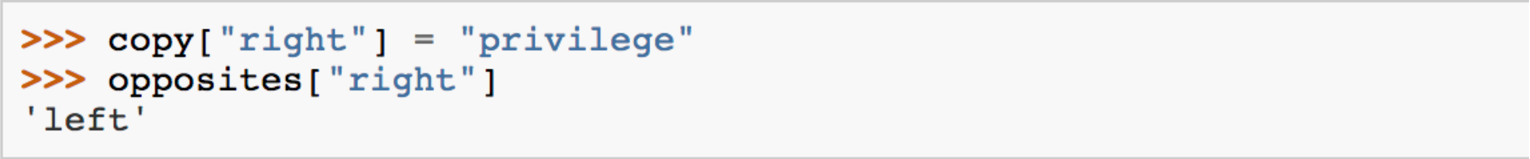
DICTIONARIES

- *Alias* and *opposites* refer to the same object;
- *Copy* refers to a fresh copy of the same dictionary.
- If *alias* is modified, *opposites* is **changed as well**:



```
>>> alias["right"] = "left"
>>> opposites["right"]
'left'
```

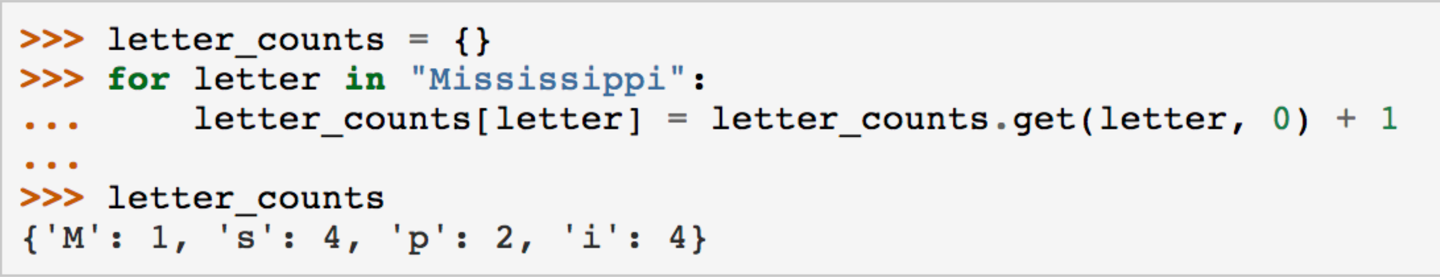
- If *copy* is modified, opposites is **unchanged**:



```
>>> copy["right"] = "privilege"
>>> opposites["right"]
'left'
```

source http://openbookproject.net/thinkcs/python/english3e/dictionaries.html

DICTIONARIES

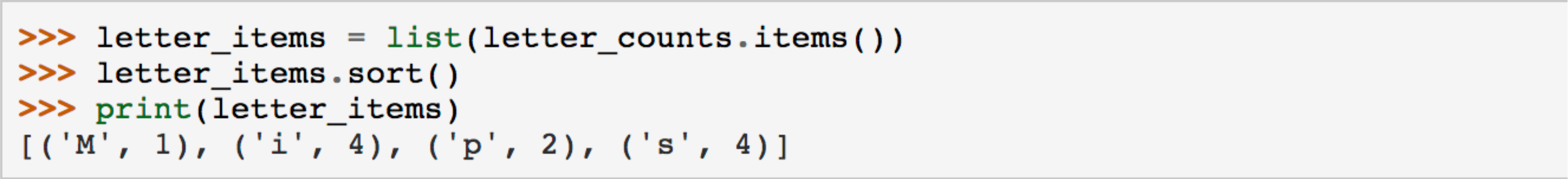


```
>>> letter_counts = {}
>>> for letter in "Mississippi":
...     letter_counts[letter] = letter_counts.get(letter, 0) + 1
...
>>> letter_counts
{'M': 1, 's': 4, 'p': 2, 'i': 4}
```

- EXAMPLE: Function that **counts the number of occurrences** of a letter in a string using a frequency table of the letters in the string (how many times each letter appears)
- Compressing a text file: *because different letters appear with different frequencies, we can compress a file by using shorter codes for common letters and longer codes for letters that appear less frequently.*
- Dictionary ideal for frequency tables

source http://openbookproject.net/thinkcs/python/english3e/dictionaries.html

DICTIONARIES



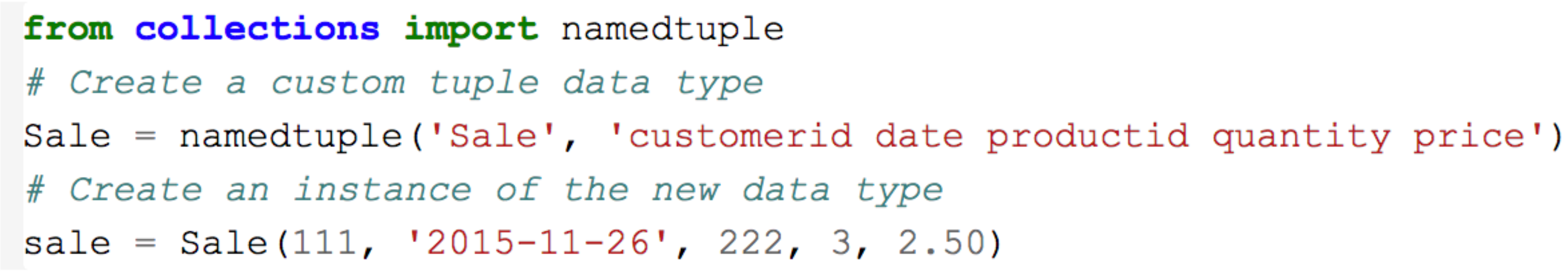
```
>>> letter_items = list(letter_counts.items())
>>> letter_items.sort()
>>> print(letter_items)
[('M', 1), ('i', 4), ('p', 2), ('s', 4)]
```

<u>ALGORITHM:</u>

- Start with an **empty dictionary**
- For each letter in the string, find the **current count** (possibly zero) and **increment it**
- At the end the dictionary contains **pairs of letters and their frequencies**
- To display the frequency table in alphabetical order use **sort()**
- <u>NOTE</u>: *in the first line the type conversion function list is called to get from items into a list (needed to use sort method)*

source http://openbookproject.net/thinkcs/python/english3e/dictionaries.html

NAMED TUPLE



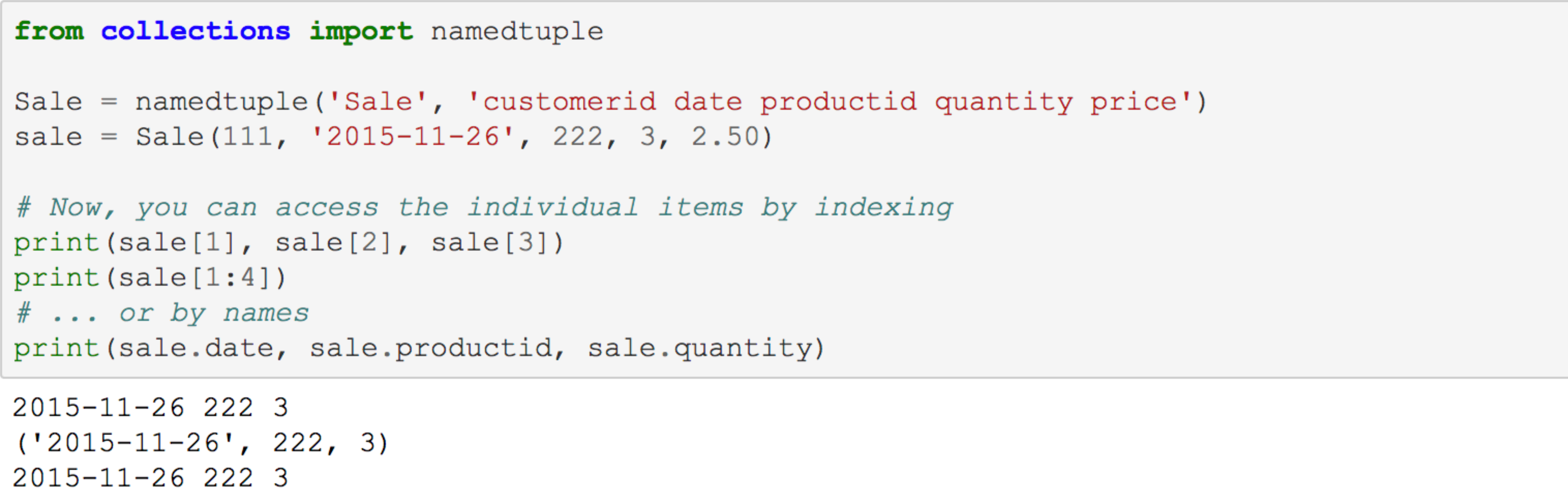
```
from collections import namedtuple
# Create a custom tuple data type
Sale = namedtuple('Sale', 'customerid date productid quantity price')
# Create an instance of the new data type
sale = Sale(111, '2015-11-26', 222, 3, 2.50)
```

- Named tuple is still a **tuple**, can be used everywhere where tuple can
- In addition to indexing by numbers it has the ability to **refer to tuple items by names**
- Other languages: struct or record
- Function **namedtuple** creates a customized tuple data type:
  - Argument 1: The **name** of the new data type
  - Argument 2: String with **space-separated names**, one for each item in our customized tuple

source courtesy of Petr Posik PRG 2016/2017

NAMED TUPLES

Named tuple: Example 1

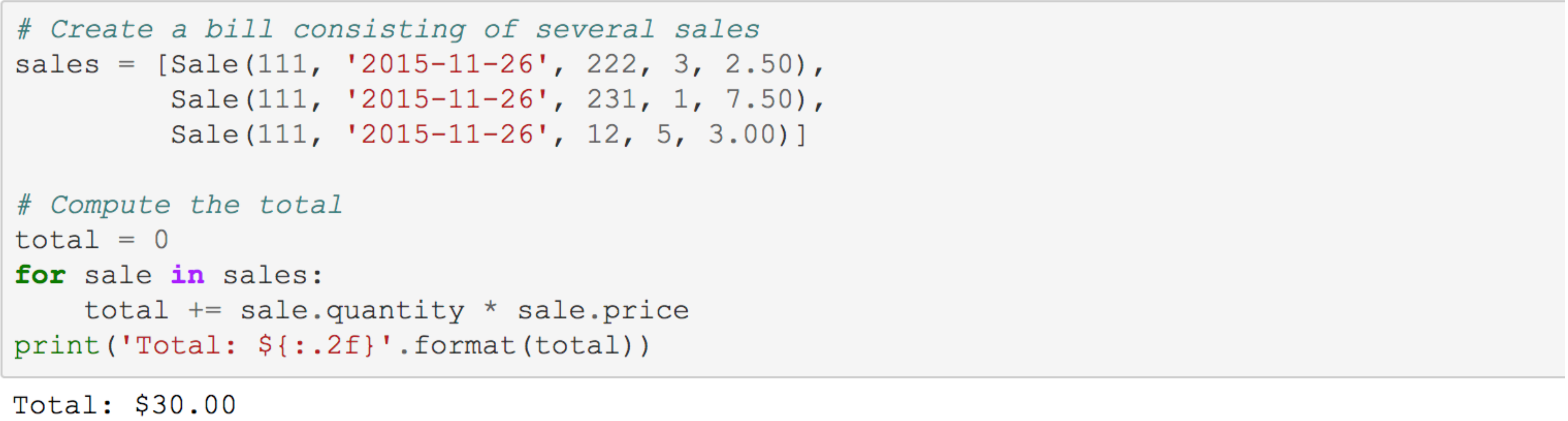


```python
from collections import namedtuple

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
```

source courtesy of Petr Posik PRG 2016/2017

NAMED TUPLES

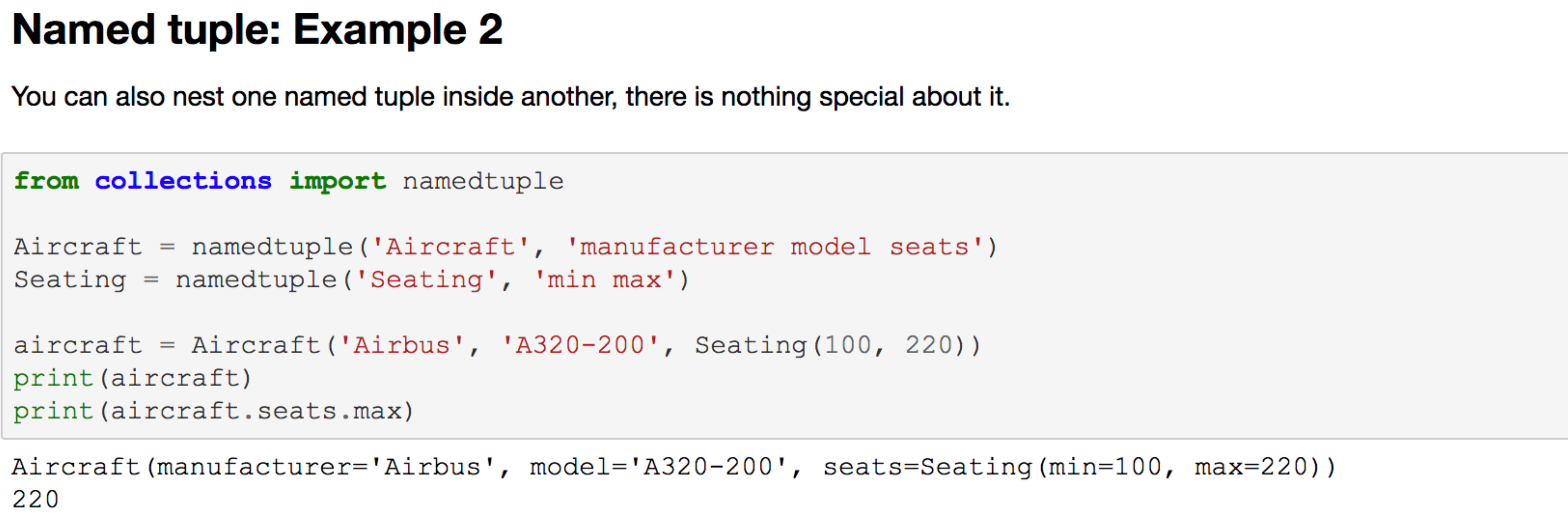


```
# Create a bill consisting of several sales
sales = [Sale(111, '2015-11-26', 222, 3, 2.50),
         Sale(111, '2015-11-26', 231, 1, 7.50),
         Sale(111, '2015-11-26', 12, 5, 3.00)]

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

```
Total: $30.00
```

source courtesy of Petr Posik PRG 2016/2017

NAMED TUPLES



Named tuple: Example 2

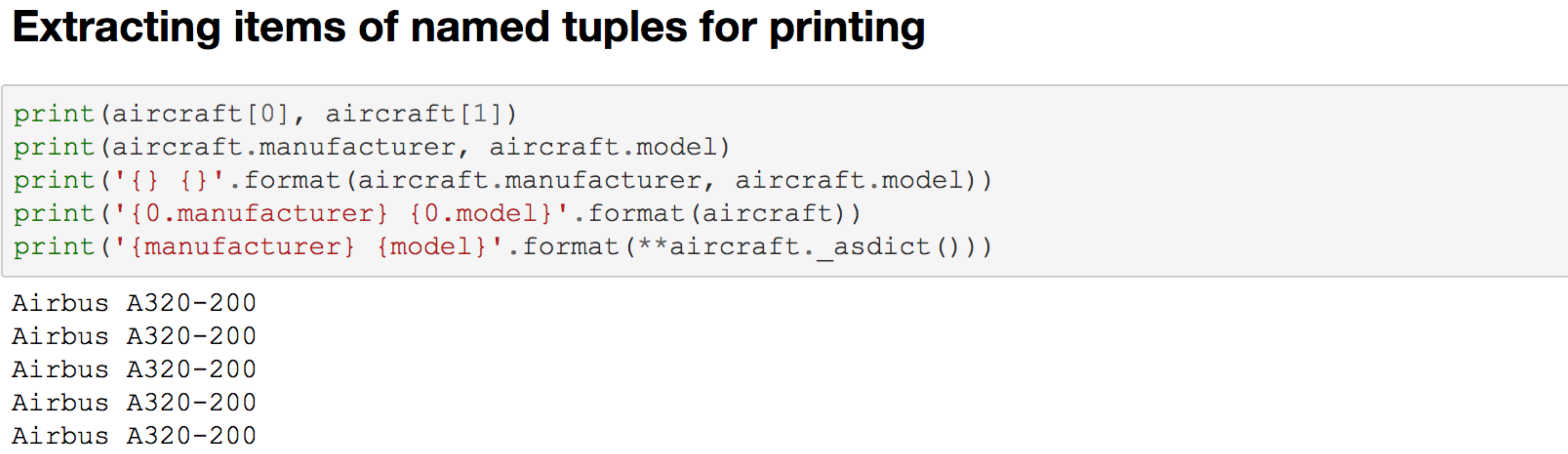
You can also nest one named tuple inside another, there is nothing special about it.

```python
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
```

source courtesy of Petr Posik PRG 2016/2017

NAMED TUPLES



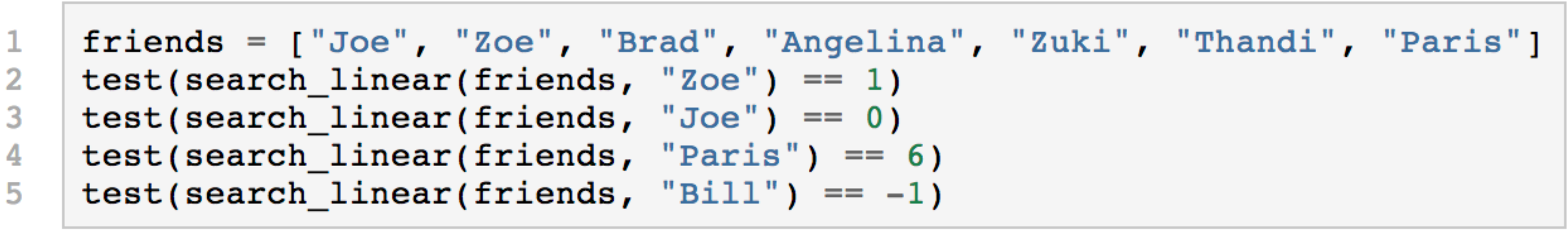
Extracting items of named tuples for printing

```
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()))
```

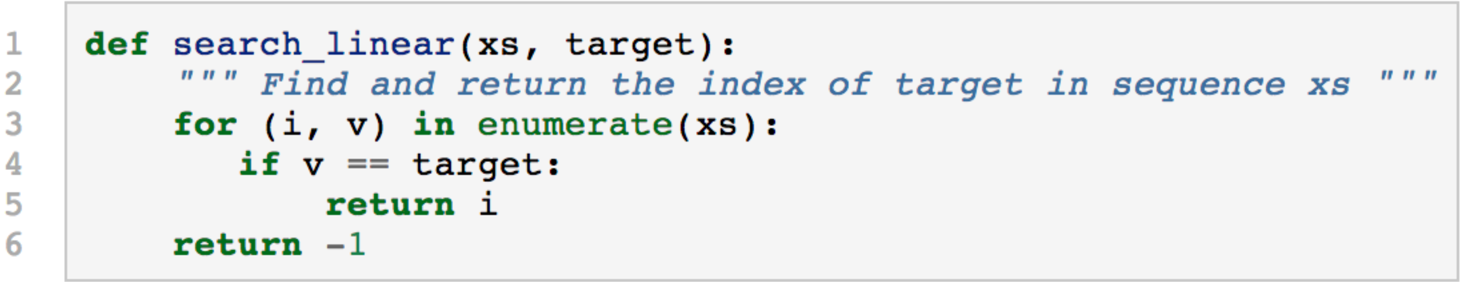
```
Airbus A320-200
Airbus A320-200
Airbus A320-200
Airbus A320-200
Airbus A320-200
```

source courtesy of Petr Posik PRG 2016/2017

LINEAR SEARCH ALGORITHM



```python
friends = ["Joe", "Zoe", "Brad", "Angelina", "Zuki", "Thandi", "Paris"]
test(search_linear(friends, "Zoe") == 1)
test(search_linear(friends, "Joe") == 0)
test(search_linear(friends, "Paris") == 6)
test(search_linear(friends, "Bill") == -1)
```

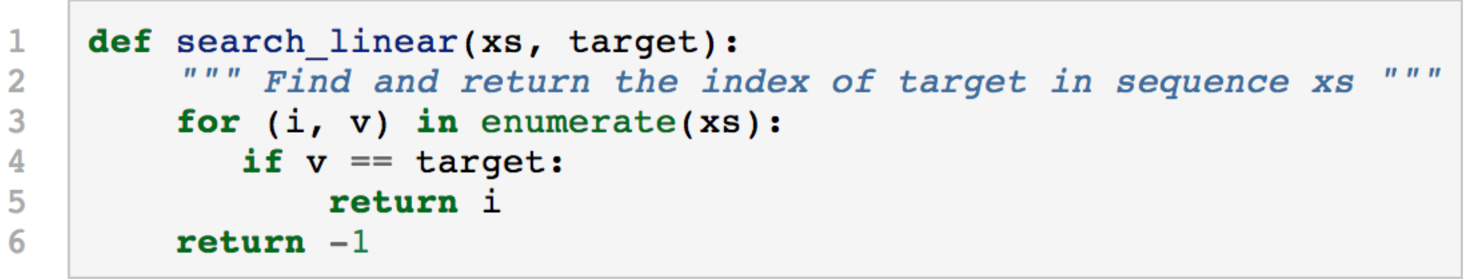


```python
def search_linear(xs, target):
    """ Find and return the index of target in sequence xs """
    for (i, v) in enumerate(xs):
        if v == target:
            return i
    return -1
```

- EXAMPLE: **Search algorithm** – *to find the index where a specific item occurs within in a list of items (return the index of the item if it is found, or return -1 if the item doesn't occur in the list)*

source http://openbookproject.net/thinkcs/python/english3e/dictionaries.html

LINEAR SEARCH ALGORITHM

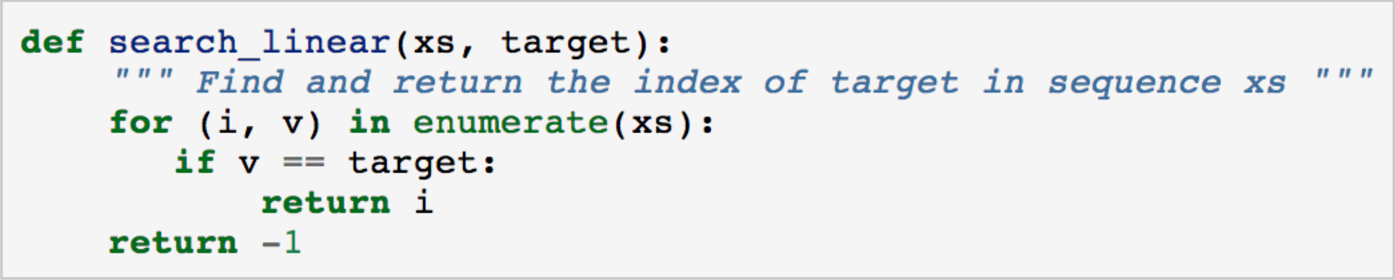


```python
def search_linear(xs, target):
    """ Find and return the index of target in sequence xs """
    for (i, v) in enumerate(xs):
        if v == target:
            return i
    return -1
```

- Searching all items in a sequence **from first to last** is called **linear search**
- Check whether **v == target** is called a **probe**
- Count probes as a measure of how **efficient** the algorithm is (*indication of how long the algorithm will take to execute*)
- Linear searching is characterized by the fact that *the number of probes needed to find some target depends directly on the length of the list*

source http://openbookproject.net/thinkcs/python/english3e/dictionaries.html

LINEAR SEARCH ALGORITHM



```python
def search_linear(xs, target):
    """ Find and return the index of target in sequence xs """
    for (i, v) in enumerate(xs):
       if v == target:
           return i
    return -1
```

- Test every item in the list from first to last such that the result is returned by the function as it is found (**early return**)
- NEGATIVE: *If searching for a target that is not present in the list, then go all the way to the end before we can return the negative value*
- Search has **linear** performance
- Interested in the scalability of our algorithms (*million or ten million items?*)

source http://openbookproject.net/thinkcs/python/english3e/dictionaries.html

REFERENCES

This lecture re-uses selected parts of the OPEN BOOK PROJECT

Learning with Python 3 (RLE)

http://openbookproject.net/thinkcs/python/english3e/index.html

available under GNU Free Documentation License Version 1.3)

- Version date: October 2012
- by Peter Wentworth, Jeffrey Elkner, Allen B. Downey, and Chris Meyers (based on 2nd edition by Jeffrey Elkner, Allen B. Downey, and Chris Meyers)
- Source repository is at https://code.launchpad.net/~thinkcspy-rle-team/thinkcspy/thinkcspy3-rle
- For offline use, download a zip file of the html or a pdf version from http://www.ict.ru.ac.za/Resources/cspw/thinkcspy3/