Lecture 5 – Collections, Sets, Dictionaries

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

Tomas Jenicek

Czech Technical University in Prague,
Faculty of Electrical Engineering, Dept. of Cybernetics,
Center for Machine Perception
http://cmp.felk.cvut.cz/~jenicto2/
tomas.jenicek@fel.cvut.cz

RECAP: MORE ABOUT PYTHON

- Everything in Python is **object**
- Python is **dynamically typed** language

- The methods and variables are created on the **stack memory**
- The objects and instances are created on the **heap memory**
- New **stack frame** is created *on invocation of a function / method* and references are assigned & counted

- Stack frames are destroyed as soon as the function / method returns
- Mechanism to clean up the dead objects is **Garbage collector** (*algorithm used is **Reference Counting** and immediate object removal if count == 0*)

source https://www.youtube.com/watch?v=arxWaw-E8QQ&t=1s

```
>>> my_string = "TEST"
>>> my_string[2] = "X"
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

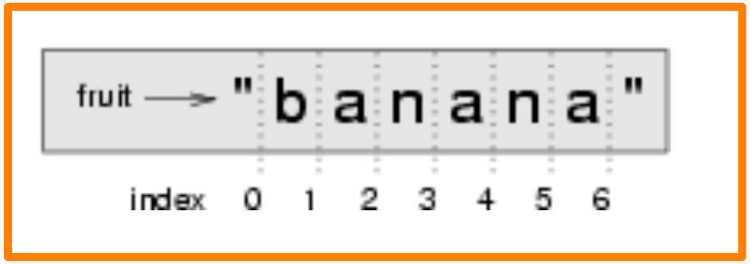
```
>>> my_list = ["T", "E", "S", "T"]
>>> my_list[2] = "X"
>>> my_list
['T', 'E', 'X', 'T']
```

- Lists are **mutable** (we can change their elements)
- Strings are **immutable** (we cannot change their elements)
- Use **slicing principles** (indexes in between characters / items)

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

RECAP: SLICING

```
>>> s = "Pirates of the Caribbean"
>>> print(s[0:7])
Pirates
>>> print(s[11:14])
the
>>> print(s[15:24])
Caribbean
>>> friends = ["Joe", "Zoe", "Brad", "Angelina", "Zuki", "Thandi", "Paris"]
>>> print(friends[2:4])
['Brad', 'Angelina']
```



- A **substring** of a string is obtained by taking a **slice**
- Slice a list to refer to some **sublist** of the items in the list
- The operator **[n:m]** returns the part of the string from the *n'th* character to the *m'th* character, **including the first but excluding the last** (*indices pointing between the characters*)
- Slice operator [n:m] **copies** out the part of the paper between the **n** and **m** positions
- Result of **[n:m]** will be of **length (m-n)**

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

RECAP: STRINGS vs. LISTS

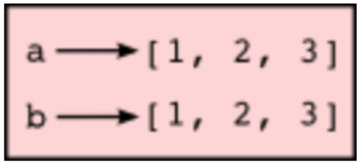
Strings

```
a = "banana"
b = "banana"
```


```
>>> a is b
True
```

Lists

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a == b
True
>>> a is b
False
```

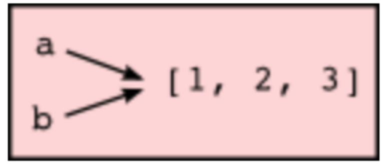


- Variables **a** and **b** refer to string object with letters "banana”
- Use **is** operator or **id** function to find out the **reference**
- Strings are **immutable**:
  Python optimizes resources by making two names that refer to the same string value refer to the same object
- Not the case of lists: **a** and **b** have the same value (content) but do not refer to the same object

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

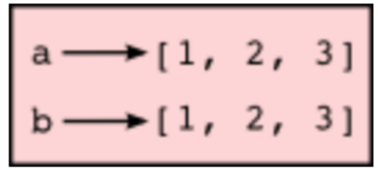
RECAP: LISTS – ALIASING, CLONING

```
>>> a = [1, 2, 3]
>>> b = a
>>> a is b
True
```




```
>>> b[0] = 5
>>> a
[5, 2, 3]
```

```
>>> a = [1, 2, 3]
>>> b = a[:]
>>> b
[1, 2, 3]
```




```
>>> b[0] = 5
>>> a
[1, 2, 3]
```

- If we assign one variable to another, both variables refer to the same object
- The **same list has two different names** we say that it is **aliased** (*changes made with one alias affect the other*)
- Recommendation is to **avoid aliasing**
- If need to modify a list and keep a copy of the original use the **slice operator** (*taking any slice of a creates a new list*)

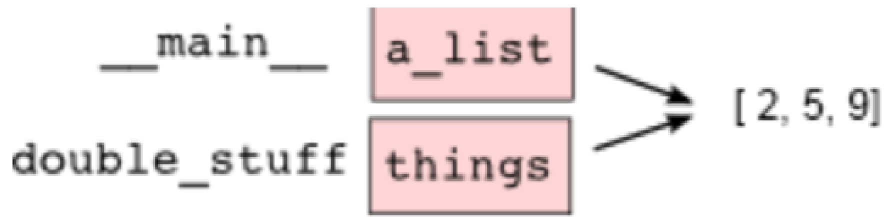
source http://openbookproject.net/thinkcs/python/english3e/lists.html

RECAP: LIST PARAMETERS


```python
def double_stuff(a_list):
    """ Overwrite each element in a_list with double its value. """
    for (idx, val) in enumerate(a_list):
        a_list[idx] = 2 * val
```

```python
things = [2, 5, 9]
double_stuff(things)
print(things)
```

```
[4, 10, 18]
```



- Passing a **list as an argument** passes a **reference** to the list, **not a copy or clone** of the list
- So **parameter passing** creates an **alias**
  (*one of the most common sources of error*)

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

RECAP: LIST PARAMETERS

```python
def double_stuff(a_list):
    """ Return a new list which contains
        doubles of the elements in a_list.
    """
    new_list = []
    for value in a_list:
        new_elem = 2 * value
        new_list.append(new_elem)

    return new_list
```

```python
def double_stuff(a_list):
    """ Overwrite each element in a_list with double its value. """
    for (idx, val) in enumerate(a_list):
        a_list[idx] = 2 * val
```

- Concept: **pure functions** vs. **modifiers**
- Pure function does not produce **side effects**!
- Pure function communicates with the calling program **only through parameters** (*it does not modify*) and a **return value**
- *Do not alter the input parameters unless really necessary*
- Programs that use pure functions are **faster to develop** and **less error-prone** than programs that use modifiers

Source by Tomas Svoboda PRG 2016/2017

functions that produce lists

- `def fcn(par):`
- `initialize result as empty list`
- `loop`
  - `create a new element`
  - `add to the result`
- `return result`

Source by Tomas Svoboda PRG 2016/2017

SEQUENCE TYPES

- Sequences of items support the following operations:
  - *membership operator* **in**
  - *querying for size / number of items* **len**
  - *indexing and slicing* **[]**
  - *are iterables**

- **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 the following operations:
  - *membership operator* **in**
  - *querying for size* **len**
  - *are iterable*
  - set operations (*comparisons, union, intersection, subset*)
- **Set**: **mutable** unordered collection of unique items of any type
- **Frozen set**: **immutable** unordered collection of unique items of any data type

source courtesy of Petr Posik BE5b33PR 2016/2017

SET TYPES

- Set types when iterated over provide items in an **arbitrary order**

- Only **hashable objects** may be added to a set:

  - Immutable data types are **hashable** (hash value does not change, objects 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

HASHABLE – THE DEFINITION

- An object is **hashable** if it **has a hash value which never changes during its lifetime**: it needs a __hash__() method and to be compared to other objects it needs an __eq__() method

- Hashable objects *which compare equal* must have *the same hash value*

- Hashability makes an object **usable as a dictionary key** and **a set member**, because *these data structures use the hash value internally*

- All of Python's **immutable built-in objects are hashable**;
  mutable containers (such as lists or dictionaries) are not hashable

- Objects which are **instances of user-defined classes are hashable by default**. They all compare *unequal* (except with themselves) and their hash value is derived from their id()

source https://docs.python.org/3/glossary.html

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

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('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: (not in both sets)

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

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

source courtesy of Petr Posik BE5b33PR 2016/2017

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 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 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 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 / number of items* **len**
  - *are 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

```
>>> 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

- 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

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

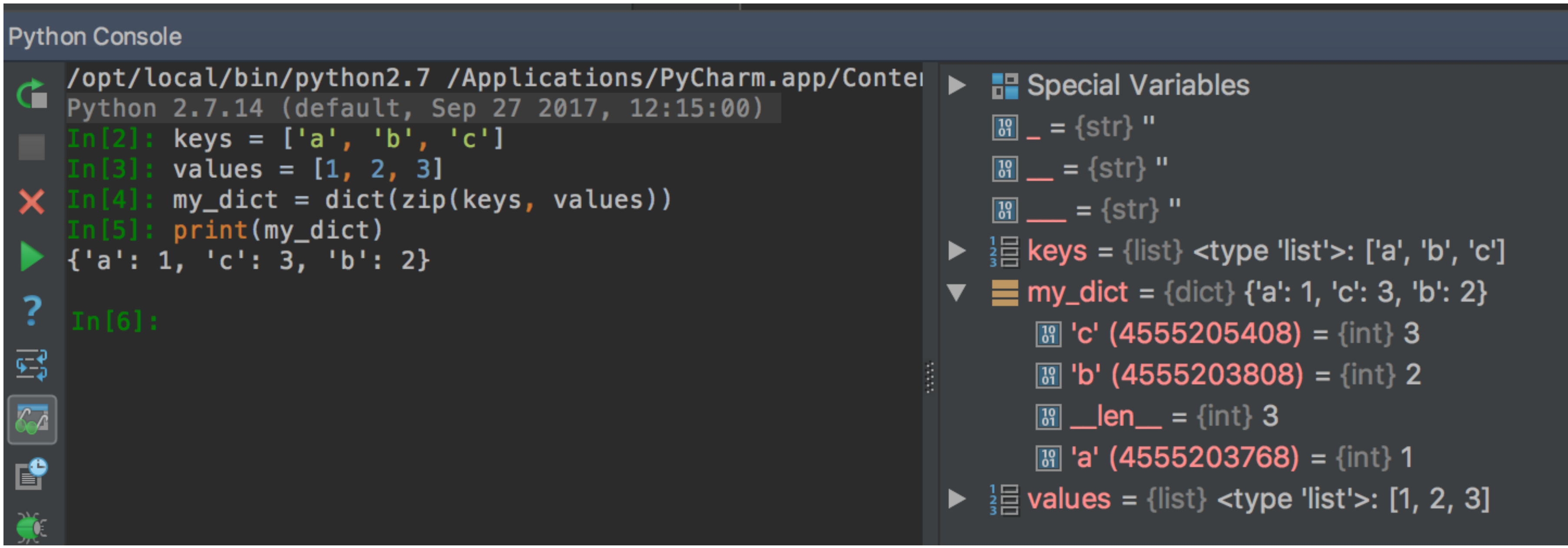
```
>>> eng2sp = {"one": "uno", "two": "dos", "three": "tres"}
```

```
>>> 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: EXAMPLE: *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



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

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

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

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

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

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

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
- 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

```
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

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

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

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`, ...).

`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

- As in the case of lists, because **dictionaries are mutable**, we need to be aware of **aliasing** (!!)

- 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

- *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

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

ALGORITHM:

- 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()**
- NOTE: *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

```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
```

- <u>EXAMPLE</u>: **Search algorithm** – *to find the index where a specific item occurs within in a list of items then 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**)
- <u>NEGATIVE</u>: *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
  (how to solve this for million or ten million of items?)

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

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/