# A Short and Incomplete Introduction to Python

## Part 7: `dicts` and other data structures

**Riccardo Murri** <riccardo.murri@uzh.ch>,
Sergio Maffioletti <sergio.maffioletti@uzh.ch>
S3IT: Services and Support for Science IT,
University of Zurich

June 7–8, 2018

# Dictionaries

# Dictionaries

The `dict` type implements a key/value mapping:

```
>>> D = { }
>>> D['a'] = 1
>>> D[2] = 'b'
>>> D
{'a': 1, 2: 'b'}
```

Dictionaries can be created and initialized using the following syntax:

```
>>> D = { 'a':1, 2:'b' }
>>> D['a']
1
```

The `for` statement can be used to loop over keys of a dictionary:

```
>>> D = { 'a':1, 'b':2 }
>>> for val in D.keys():
...     print(val)
'a'
'b'
```

Loop over dictionary *keys*. *The* `.keys()` *part can be omitted, as it's the default!*

If you want to loop over dictionary *values*, you have to explicitly request it.

```
>>> D = dict(a=1, b=2)
>>> for val in D.values():
...     print(val)
1
2
```

Loop over dictionary *values*
*The* `.values()` *cannot be omitted!*

# The 'in' operator (2)

Use the **in** operator to test for presence of an item in a collection.

**x in D**

**x in D.keys()**

Evaluates to `True` if `x` is equal to a *key* in the `D` dictionary.

**x in D.values()**

Evaluates to `True` if `x` is equal to a *value* in the `D` dictionary.

**Exercise 7.A:** Write a function `wordcount(filename)` that reads a text file and returns a dictionary, mapping words into occurrences (disregarding case) of that word in the text.

For example, using the lorem_ipsum.txt file:

```
>>> wordcount('lipsum.txt')
{'and': 3, 'model': 1, 'more-or-less': 1,
 'letters': 1, [...]
```

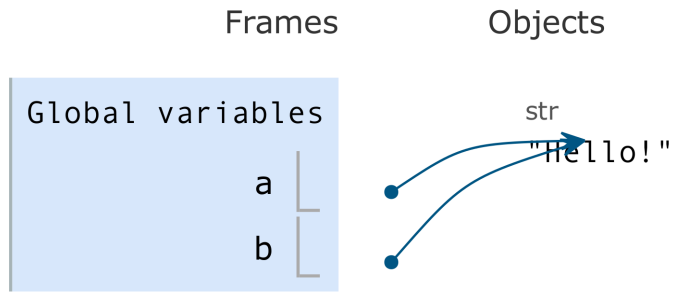
For the purposes of this exercise, a "word" is defined as a sequence of letters and the character "-", i.e., "e-mail" and "more-or-less" should both be counted as a single word.

# Variables are names for values

**Python variables are just "names" given to values.**

This allows you to *reference* the string `'Python'` by the *name* `a`. But also by another name `b`:

```
1  a = "Hello!"
2  b = a
```




The *same* object can be given many names!

*See also:* http://excess.org/article/2014/04/bar-foo/

# The `is` operator

The `is` operator allows you to test whether two names refer to the same object:

```
>>> a = 1
>>> b = 1
>>> a == b
True
>>> a is b
True
```

```
>>> a = ''
>>> b = ""
>>> a == b
True
>>> a is b
True
```

```
>>> a = list()
>>> b = list()
>>> a == b
True
>>> a is b
False
```

# All variables are references

In Python, **all objects are ever passed by reference**.

In particular, **variables always store a reference to an object**, never a copy!

Hence, you have to be careful when modifying objects:

```
>>> a = [1,2,3]
>>> b = a
>>> b.remove(2)
>>> print(a)
???
```

**Q**: How many items are in the `a` list now?

# All variables are references

In Python, **all objects are ever passed by reference**.

In particular, **variables always store a reference to an object**, never a copy!

Hence, you have to be careful when modifying objects:

```
>>> a = [1,2,3]
>>> b = a
>>> b.remove(2)
>>> print(a)
[1, 3]
```

Run this example in the Online Python Tutor to better understand what's going on.

*This applies particularly for variables that capture the arguments to a function call!*

# All variables are references (demo)

www.pythontutor.com

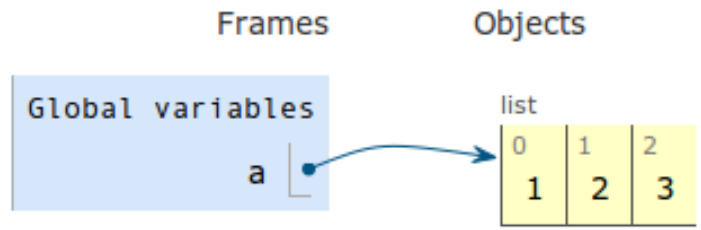
```
1  a = [1, 2, 3]
2  b = a
3  b.remove(2)
4  print a
5  print b
```

Frames Objects

# All variables are references (demo)

www.pythontutor.com

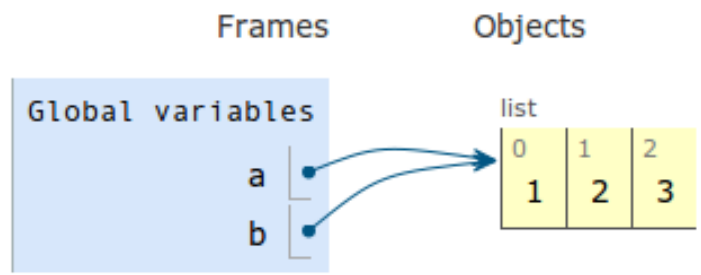
```
1  a = [1, 2, 3]
2  b = a
3  b.remove(2)
4  print a
5  print b
```



# All variables are references (demo)

www.pythontutor.com

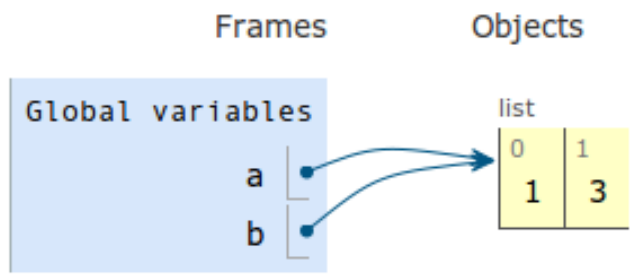
```
1  a = [1, 2, 3]
2  b = a
3  b.remove(2)
4  print a
5  print b
```



# All variables are references (demo)

www.pythontutor.com

```
1  a = [1, 2, 3]
2  b = a
3  b.remove(2)
4  print a
5  print b
```



# How to copy an object? (0)

```
>>> a = [1, 2]
>>> b = a
>>> b.remove(1)
>>> print(b)
[2]
>>> print(a)
[2]
```

# How to copy an object? (1)

```
>>> from copy import copy
>>> a = [1, 2]
>>> b = copy(a)
>>> b.remove(1)
>>> print(b)
[2]
>>> print(a)
[1, 2]
```

# How to copy an object? (2)

Note that `copy.copy` makes a *shallow* copy:

```
>>> D = { 'a':[1,2], 'b':3 }
>>> print(D['a'])
[1, 2]
>>> E = copy(D)
>>> print(E)
{ 'a':[1, 2], 'b':3 }
>>> E['a'].remove(1)
>>> print(D['a'])
[2]
```

# How to copy an object? (3)

To make a copy of nested data structures, you need `copy.deepcopy`:

```
>>> from copy import deepcopy
>>> D = { 'a':[1,2], 'b':3 }
>>> print(D['a'])
[1, 2]
>>> E = deepcopy(D)
>>> print(E)
{ 'a':[1, 2], 'b':3 }
>>> E['a'].remove(1)
>>> print(D['a'])
[1, 2]
>>> print(E['a'])
[2]
```

# Appendix

# Mutable vs Immutable

Some objects (e.g., `tuple`, `int`, `str`) are *immutable* and cannot be modified.

```
>>> S = 'UZH'
>>> S[2] = 'G'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignmen
```

`list`, `dict`, `set` and user-defined objects are *mutable* and can be modified in-place.

# Dictionary, sets and mutable objects

Not all objects can be used as dictionary *keys* or items in a set:

- *Immutable* objects **can be** used as `dict` keys or set items.
- *Mutable* objects **cannot be** used as `dict` keys or set items.

(Explanation for the technically savvy: a dictionary is essentially a Hash Table, therefore keys of a dictionary must be *hashable* objects. If objects were allowed to mutate, their hash value would change too and we would lose the mapping.)