A Short and Incomplete Introduction to Python

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

March 2, 2017

Welcome!

What is S3IT?

"A partner for data- and compute-intensive science"

Enable researchers and projects to run simulations and data analysis.

Develop tools to integrate, automate and scale scientific use cases.

Provide access to *innovative* infrastructures and technologies.

Want to know more? http://www.s3it.uzh.ch

Prerequisites

This course assumes a basic experience with computer programming.

Any language should do, as long as you are already familiar with the concepts of variables and functions.

Python 2 *vs* Python 3

There are currently two major versions of Python available, with slightly different syntax and features.

Python 2.7 is the last release in the 2.x series.

Python 3.x has a more polished syntax, removing inconsistencies and some historical baggage.

But Python 2.x is still the default on most Linux distributions, so **we shall focus on Py2 syntax**.

Watch a debate between "Pro" and "Contra" advocates:
http://www.physik.uzh.ch/~nchiapol/webm/3_1_Python3.webm

Explore the key differences:
http://tinyurl.com/py2-and-py3-key-differences

Talk outline

1. Python basics
2. NumPy and plotting
3. Pandas and how to query tabular data

Next steps

The course will be structured as a mixture of slides and hands-on sessions for practicing Python programming.

So, the very first step is making sure you can access the Jupyter/IPython server for running the exercise notebooks.

Python basics

The Python shell, I

Python is an *interpreted* language.

It also features an interactive "shell" for evaluating expressions and statements immediately.

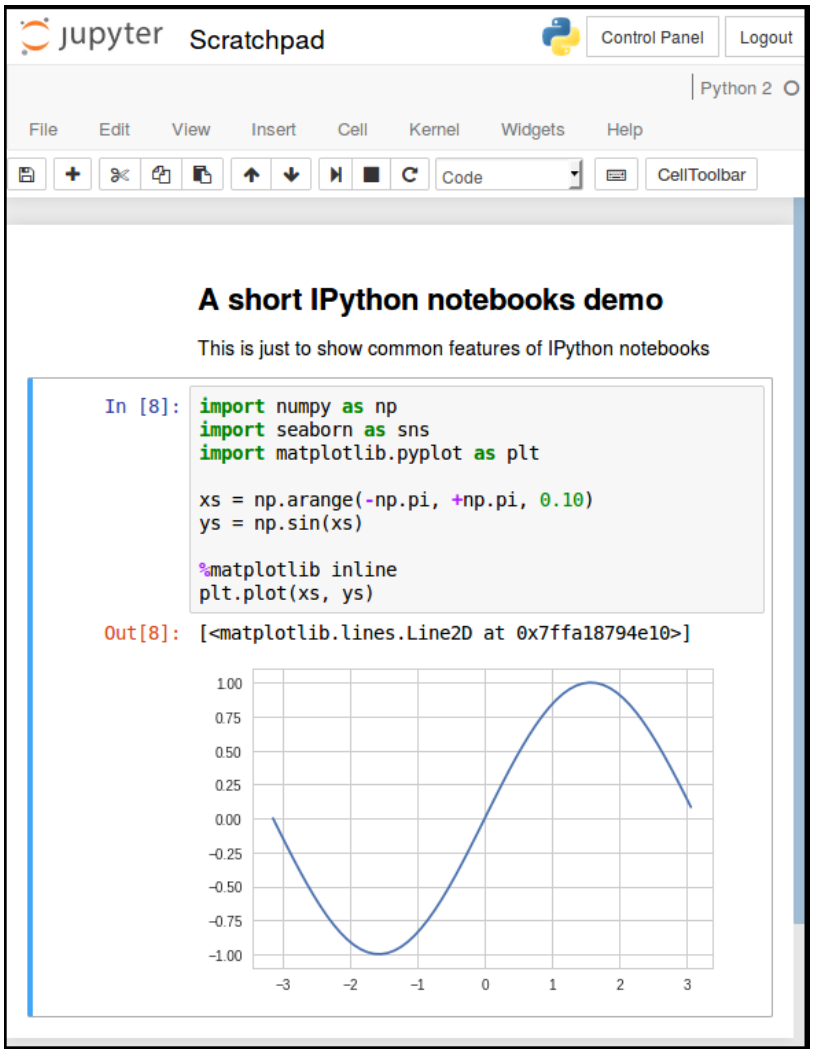
The IPython shell is started by invoking the command `ipython` in a terminal window.

```
$ ipython
Python 2.7.13 |Anaconda 4.3.0 (64-bit)| (default, Dec 20 2016, 23:09:15)
Type "copyright", "credits" or "license" for more information.

IPython 5.1.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: ← here is where you enter commands
```

The IPython notebook, I



A more appealing way of interacting with Python is through the IPython notebooks.

Notebooks are made of "cells", which come in two flavors:

- documentation cells, containing text formatted according to the Markdown conventions;
- code cells, containing arbitrary Python code

The IPython notebook, II

To run Python code in the notebook:

- Type your code in a cell besides the `In [ ]:` (multiple lines are allowed)
- Press **Ctrl+Enter** to evaluate the cell (prompt changes to `In [*]:`) — or press **Alt+Enter** to evaluate the code *and* open a new code cell.
- When the Python kernel has done computing, the result appears *under* the code cell marked with a `Out [ ]:` label.

The Python shell, II

Expressions can be entered at the Python shell prompt (the '>>>' at the start of a line); they are evaluated and the result is printed:

```
>>> 2+2
4
```

Throughout these slides, all Python code marked with '>>>' can also be entered and evaluated in the IPython notebook cells:

```
In [1]: 2+2
Out[1]: 4
```

Lines of Python code

Line of Python code are ended by the "new line" character. (I.e., when you press the *Enter* key.)

A line can be continued onto the next by ending it with the character '\'; for example:

```
>>> "hello" + \
... " world!"
'hello world!'
```

The prompt changes to '. . .' on continuation lines.

Reference:

http://docs.python.org/reference/lexical_analysis.html#line-structure

String literals, I

There are several ways to express string literals in Python.

Single and double quotes can be used interchangeably:

```
>>> "a string" == 'a string'
True
```

You can use the single quotes inside double-quoted strings, and viceversa:

```
>>> a = "Isn't it ok?"
>>> b = '"Yes", he said.'
```

String literals, II

Multi-line strings are delimited by three quote characters.

```
>>> a = """This is a string,
... that extends over more
... than one line.
... """
```

In other words, you need not use the backslashes "\" at the end of the lines.

Operators

All the usual unary and binary arithmetic operators are defined in Python: `+`, `-`, `*`, `/`, `**` (exponentiation), `<<`, `>>`, etc.

Logical operators are expressed using plain English words: `and`, `or`, `not`.

Numerical and string comparison also follows the usual notation: `<`, `>`, `<=`, `==`, `!=`, ...

Reference:

- http://docs.python.org/library/stdtypes.html#boolean-operations-and-or-not
- http://docs.python.org/library/stdtypes.html#comparisons

Your first exercise

How much is 2^{38} ?

(You have 1 minute time.)

Operators, II

Some operators are defined for non-numeric types:

```
>>> "U" + 'ZH'
'UZH'
```

Some support operands of mixed type:

```
>>> "a" * 2
'aa'
>>> 2 * "a"
'aa'
```

Some do not:

```
>>> "aaa" / 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

String interpolation

The `.format()` method can be used to substitute values into placeholder strings.

Placeholders can indicate substitutions by ordinal number:

```
>>> "This is slide {0} of {1}".format(20, 1001)
'This is slide 20 of 1001.'
```

You can use names instead of numbers (then the order parameter occur in `format()` does not matter):

```
>>> "Today is {month} {day}".format(day=2, month='March')
'Today is March 2'
```

Reference: https://pyformat.info/

String interpolation

The `.format()` method can be used to substitute values into placeholder strings.

Placeholders can indicate substitutions by ordinal number:

```
>>> "This is slide {0} of {1}".format(20, 1001)
'This is slide 20 of 1001.'
```

You can use names instead of numbers (then the order parameter occur in `format()` does not matter):

```
>>> "Today is {month} {day}".format(day=2, month='March')
'Today is March 2'
```

Reference: https://pyformat.info/

String interpolation

The `.format()` method can be used to substitute values into placeholder strings.

Placeholders can indicate substitutions by ordinal number:

```
>>> "This is slide {0} of {1}".format(20, 1001)
'This is slide 20 of 1001.'
```

You can use names instead of numbers (then the order parameter occur in `format()` does not matter):

```
>>> "Today is {month} {day}".format(day=2, month='March')
'Today is March 2'
```

Reference: https://pyformat.info/

Assignment, I

Assignment is done via the '=' statement:

```
>>> a = 1
>>> print(a)
1
```

There are a few shortcut notations:

- `a += b` short for `a = a + b`,
- `a -= b` short for `a = a - b`,
- `a *= b` short for `a = a * b`,

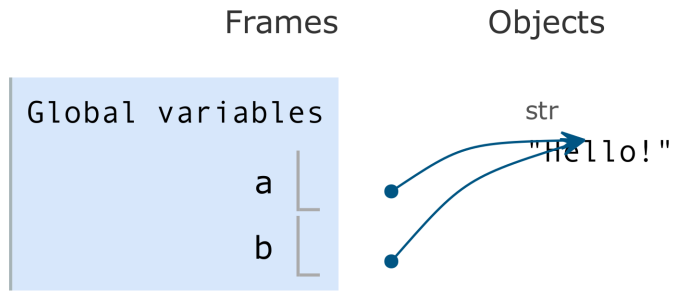
etc. — one for every legal operator.

Assignment, II

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 is b
True
```

Functions, I

Functions are called by postfixing the function name with a parenthesized argument list.

```
>>> int("42")
42
>>> int(4.2)
4
>>> float(42)
42.0
>>> str(42)
'42'
>>> str()
''
```

Functions, II

Some functions can take a variable number of arguments. For instance:

sum($x_0, \ldots, x_n$) Return $x_0 + \cdots + x_n$.

max($x_0, \ldots, x_n$) Return the maximum of $\{x_0, \ldots, x_n\}$

min($x_0, \ldots, x_n$) Return the minimum of $\{x_0, \ldots, x_n\}$

Examples:

```
>>> min(1,2,3)
1
>>> max(1,2)
2
```

The most important function of all

help(`fn`) Display help on the function named `fn`

Q: *What happens if you type these at the prompt?*

- *`help(abs)`*
- *`help(help)`*

How to define new functions

The **def** statement starts a function definition.

```
def greet(name):
    """
    A friendly function.
    """
    print ("Hello, " + name + "!")

# the customary greeting
greet("world")
```

Indentation is significant in Python: it is used to delimit blocks of code, like '{' and '}' in Java and C.

```
def greet(name):
    """
    A friendly function.
    """
    print ("Hello, " + name + "!")

# the customary greeting
greet("world")
```

(This is a comment. It is ignored by Python, just like blank lines.)

```
def greet(name):
    """
    A friendly function.
    """
    print ("Hello, " + name + "!")

# the customary greeting
greet("world")
```

This calls the function just defined.

```
def greet(name):
    """
    A friendly function.
    """
    print ("Hello, " + name + "!")

# the customary greeting
greet("world")
```

What is this? The answer in the next exercise!

```
def greet(name):
    """
    A friendly function.
    """
    print ("Hello, " + name + "!")

# the customary greeting
greet("world")
```

Exercise 3: Type and run the code on the previous page at the interactive prompt. (Pay attention to indentation!)

What's the result of evaluating the function `greet("world")`?

What does `help(greet)` output?

Default values

Function arguments can have default values.

```
>>> def hello(name='world'):
...     print ("Hello, " + name)
...
>>> hello()
'Hello, world'
```

Named arguments

Python allows calling a function with named arguments:

```
hello(name="Alice")
```

When passing arguments by name, they can be passed in any order:

```
>>> from fractions import Fraction
>>> Fraction(numerator=1, denominator=2)
Fraction(1, 2)
>>> Fraction(denominator=2, numerator=1)
Fraction(1, 2)
```

The 'return' statement

```
def double(x):
    return x+x

double(3) == 6
```

The result of a function evaluation is set by the *return* statement.

If no `return` is present, the function returns the special value `None`.

```
def double(x):
    return x+x
    # the following line
    # is never exec'd
    print('Hello')
```

After executing `return` the control flow leaves the function.

Conditionals

Conditional execution uses the `if` statement:

```
if expr:
  # indented block
elif other-expr:
  # indented block
else:
  # executed if none of the above matched
```

The `elif` can be repeated, with different conditions, or left out entirely.

Also the `else` clause is optional.

Q: *Where's the 'end if'?*

Conditionals

Conditional execution uses the `if` statement:

```
if expr:
   # indented block
elif other-expr:
   # indented block
else:
   # executed if none of the above matched
```

The `elif` can be repeated, with different conditions, or left out entirely.

Also the `else` clause is optional.

> **Q:** *Where's the 'end if'?*
> *There's no 'end if': indentation delimits blocks!*

`while`-Loops

Conditional looping uses the `while` statement:

`while` *expr*:
```
  # indented block
```

To break out of a `while` loop, use the `break` statement.

Use the `continue` statement anywhere in the indented block to jump back to the `while` statement.

Exercise 4: Modify the `greet()` function to print "Welcome back!" if the argument `name` is the string `'BrainHack'`.

Modules, I

The `import` statement reads a `.py` file, executes it, and makes its contents available to the current program.

```
>>> import hello
Hello, world!
```

Modules are only read once, no matter how many times an `import` statement is issued.

```
>>> import hello
Hello, world!
>>> import hello
>>> import hello
```

Modules, II

Modules are *namespaces:* functions and variables defined in a module must be prefixed with the module name when used in other modules:

```
>>> hello.greet("Python")
Hello, Python!
```

To import definitions into the current namespace, use the '`from` *x* `import` *y*' form:

```
>>> from hello import greet
>>> greet("Python")
Hello, Python!
```

Sequences

Python provides a few built-in *sequence* classes:

list	*mutable*, possibly heterogeneous
tuple	*immutable*, possibly heterogeneous
str	*immutable*, only holds characters

Additional sequence types are provided by external modules:

array	*mutable*, homogeneous (like C/Fortran arrays, from NumPy)
DataFrame	*mutable*, heterogeneous (like R, from Pandas)

Lists - *(mutable, heterogeneous)*

Lists are by far the most common and used sequence type in Python.

Lists are created and initialized by enclosing values into '[' and ']':

```
>>> L = [ 'U', 'Z' ]
```

You can append and remove items from a list:

```
>>> L.append('H')
>>> print (L)
['U', 'Z', 'H']
```

You can append **any** object to a list:

```
>>> L.append([1, 2])
>>> print(L)
['U', 'Z', 'H', [1, 2]]
```

Lists - *(mutable, heterogeneous)*

Lists are by far the most common and used sequence type in Python.

Lists are created and initialized by enclosing values into '[' and ']':

```
>>> L = [ 'U', 'Z' ]
```

You can append and remove items from a list:

```
>>> L.append('H')
>>> print (L)
['U', 'Z', 'H']
```

You can append **any** object to a list:

```
>>> L.append([1, 2])
>>> print(L)
['U', 'Z', 'H', [1, 2]]
```

Lists - *(mutable, heterogeneous)*

Lists are by far the most common and used sequence type in Python.

Lists are created and initialized by enclosing values into '[' and ']':

```
>>> L = [ 'U', 'Z' ]
```

You can append and remove items from a list:

```
>>> L.append('H')
>>> print (L)
['U', 'Z', 'H']
```

You can append **any** object to a list:

```
>>> L.append([1, 2])
>>> print(L)
['U', 'Z', 'H', [1, 2]]
```

Sequences, II

You can access individual items in a sequence using the postfix `[]` operator.

Sequence indices start at 0.

```
>>> L = ['U', 'Z', 'H']
>>> print(L[0], L[1], L[2])
'U' 'Z' 'H'
>>> S = 'UZH'
>>> print(S[0], S[1], S[2])
'U' 'Z' 'H'
```

Sequence length

The `len()` function returns the number of items in any sequence (not just lists).

```
>>> len(L)
3
```

Built-in functions **`sum()`**, **`max()`**, **`min()`** also work on list arguments.

Exercise 6: Write a function `avg()` that takes a list of numbers and returns their mean value.

Slices

The notation `[n:m]` is used for accessing a *slice* of sequence (the items at positions n, $n+1$, ..., $m-1$).

```
>>> # list numbers from 0 to 9
>>> R = list(range(0,10))
>>> R[1:4]
[1, 2, 3]
```

If n is omitted it defaults to 0, if m is omitted it defaults to the length of the sequence.

A *slice* of a sequence is a sequence *of the same type*.

```
>>> S = 'zurich'
>>> S[0:4]
'zuri'
```

List mutation

You can replace items in a *mutable* sequence by assigning them a new value:

```
>>> L = ['U', 'Z', 'H']
>>> L[2] = 'G'
>>> print(L)
['U', 'Z', 'G']
```

Operating on lists

Python provides a number of methods to modify a list:

L.append(x)
Append item `x` to list `L`.

L.insert(n, x)
Insert item `x` at position `n` of list `L`; other items are "shited to the right" to make place.

L.remove(x)
Remove first occurence of item having value `x` from list `L`.

L.pop(n)
Remove item at position `n` from list `L`.

Operating on lists, II

L.index(x)
Return position of first item in `L` having value `x`.

L.count(x)
Return number of items in `L` having value `x`.

L.extend(K)
Graft a list `K` to the end of list `L`.

Reference: https://docs.python.org/2/library/stdtypes.html#typesseq

Lists operators

You can concatenate two lists using the + operator:

```
>>> [1, 2] + [3, 4]
[1, 2, 3, 4]
```

You can mutate a list in place with the += operator:

```
>>> L = [1, 2]
>>> L += [3, 4]
>>> print(L)
[1, 2, 3, 4]
```

The * operator also works on lists:

```
>>> L = [1, 2]
>>> print(L*3)
[1, 2, 1, 2, 1, 2 ]
```

`for`-loops

With the `for` statement, you can **loop over the items of a sequence**:

```
for i in range(0, 4):
  # loop block
  print (i*i)
```

To break out of a `for` loop, use the `break` statement.

To jump to the next iteration of a `for` loop, use the `continue` statement.

The `for` statement can be used to loop over elements in *any sequence*.

```
>>> for val in [1,2,3]:
...   print(val)
1
2
3
```

Loop over lists

The `for` statement can be used to loop over elements in *any sequence*.

```
>>> for val in 'UZH':
...     print(val)
'U'
'Z'
'H'
```

Loop over strings

If you want to loop over a *sorted* sequence you can use the function `sorted()` :

```
>>> for val in sorted([1,3,2]):
...  print(val)
1
2
3
```

and to loop over a sequence in *inverted* order you can use the `reversed()` function:

```
>>> for val in reversed('UZH):
...     print(val)
'H'
'Z'
'U'
```

Exercise 7: Write a function `odd` that takes a list of integers and returns a list of all the odd ones.

Exercise 8: Write a function `deviation(L, m)` that takes a list `L` of numbers and a single value `m` returns a list with the difference of `m` and each element `x` of `L`.

map, reduce, filter (1)

Constructing a new list by looping over a given list and applying a function on all elements is so common that there are specialized functions for that:

map(fn, L)
Return a new list formed by applying function `fn(x)` to every element `x` of list `L`

filter(fn, L)
Return a new list formed by elements `x` of list `L` for which `fn(x)` evaluates to a "True" value.

map, reduce, filter (2)

reduce(fn2, L)

Apply function `fn2(x,y)` to the first two items `x` and `y` of list `L`, then apply `fn2` to the result and the third element of `L`, and so on until all elements have been processed — return the final result.

See also: http://www.python-course.eu/lambda.php and https://docs.python.org/3/howto/functional.html (more advanced)

This is how you could rewrite Exercises 7 and 8 using `map` and `filter`.

```
# *** Exercise 7 ***
def is_odd(x):
  return (x % 2 == 0)

def odd(L):
  return filter(is_odd, L)
```

```
# *** Exercise 8 ***
def deviation(L, m):
  # note:  can define
  # func's in func's!
  def delta(x):
    return abs(x-m)
  return map(delta, L)
```

File I/O

Code for processing a text file usually looks like this:

```
with open(filename, 'r') as stream:
  # prepare for processing
  for line in stream:
    # process each line
```

File I/O

```
with open(filename, 'r') as stream:
    # prepare for processing
    for line in stream:
        # process each line
```

The **open**(path, mode) function opens the file located at path and returns a "file object" that can be used for reading and/or writing.

Mode is one of 'r', 'w' or 'a' for reading, writing (truncates on open), appending. You can add a '+' character to enable read+write (other effects being the same).

File I/O

```
with open(filename, 'r') as stream:
  # prepare for processing
  for line in stream:
    # process each line
```

This is equivalent to `stream = open(...)` but in addition *closes* the file when the code in the `with`-block is done.

There are many more uses of the `with` statement besides automatically closing files, check out https://jeffknupp.com/blog/2016/03/07/python-with-context-managers/

File I/O

```
with open(filename, 'r') as stream:
    # prepare for processing
    for line in stream:
        # process each line
```

A `for`-loop can be used to process all lines in a file, as if the file were a list.

More on File I/O

The `.read()` method can be used to read the *whole* contents of a file in one go as a single string:

```
>>> s = stream.read()
```

Method `.readlines()` returns a list of all lines in the file:

```
>>> L = stream.readlines()
```

Reference: http://docs.python.org/library/stdtypes.html#file-objects

Type conversions

str(x) Converts the argument x to a string; for numbers, the base 10 representation is used.

int(x) Converts its argument x (a number or a string) to an integer; if x is a a floating-point literal, decimal digits are truncated.

float(x) Converts its argument x (a number or a string) to a floating-point number.

Exercise 9: Write a function `load_data(filename)` that reads a file containing one integer number per line, and return a list of the integer values.

Test it with the values.dat file:

```
>>> load_data('values.dat')
[299850, 299740, 299900, 300070, 299930]
```

Operations on strings, I

`s.capitalize()`, `s.lower()`, `s.upper()`
Return a *copy* of the string capitalized / turned all lowercase / turned all uppercase.

`s.split(t)`
Split `s` at every occurrence of `t` and return a list of parts. If `t` is omitted, split on whitespace.

`s.startswith(t)`, `s.endswith(t)`
Return `True` if `t` is the initial/final substring of `s`.

Reference: http://docs.python.org/library/stdtypes.html#string-methods

Operations on strings, II

`s.replace(old, new)`
Return a *copy* of string `s` with all occurrences of substring `old` replaced by `new`.

`s.lstrip()`, `s.rstrip()`, `s.strip()`
Return a *copy* of the string with the leading (resp. trailing, resp. leading *and* trailing) whitespace removed.

Reference: http://docs.python.org/library/stdtypes.html#string-methods

Filesystem operations, I

These functions are available from the `os` module.

`os.getcwd(), os.chdir(path)`
Return the path to the current working directory / Change the current working directory to `path`.

`os.listdir(dir)`
Return list of entries in directory `dir` (omitting '.' and '..')

`os.makedirs(path)`
Create a directory; no-op if the directory already exists. Creates all the intermediate-level directories needed to contain the leaf.

`os.rename(old,new)`
Rename a file or directory from `old` to `new`.

Reference: http://docs.python.org/library/os.html

Filesystem operations, II

These functions are available from the `os.path` module.

`os.path.exists(path)`, `os.path.isdir(path)`, `os.path.isfile(path)`
Return `True` if `path` exists / is a directory / is a regular file.

`os.path.basename(path)`, `os.path.dirname(path)`
Return the base name (the part after the last '/' character) or the directory name (the part before the last / character).

`os.path.abspath(path)`
Make `path` absolute (i.e., start with a /).

Reference: http://docs.python.org/library/os.path.html

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!*

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.)

The 'in' operator (1)

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

x in S

Evaluates to `True` if `x` is equal to a *value* contained in the `S` sequence (list, tuple, set).

x in T

Evaluates to `True` if `x` is a substring of string `T`.

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 11: 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('lorem_ipsum.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.

Exceptions

"Exceptions" is the name given in Python to error conditions.

You can write code that intercepts some error conditions and reacts appropriately.

See also: http://docs.python.org/library/exceptions.html

What does an exception look like?

```
>>> stream.write('foo')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: File not open for writing
```

What does an exception look like?

```
>>> stream.write('foo')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: File not open for writing
```

This is the exception *message*: it is supposed to be read by the (human) user.

What does an exception look like?

```
>>> stream.write('foo')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: File not open for writing
```

This is the exception *class name*; it is used for catching exceptions (syntax in the next slide).

```
try:
  # code that might raise an exception
except SomeException:
  # handle some exception
except AnotherException, ex:
  # the actual Exception instance
  # is available as variable 'ex'
finally:
  # performed on exit in any case
```

The optional **`finally`** clause is executed on exit from the **`try`** or **`except`** block in *any* case.

Reference: http://docs.python.org/reference/compound_stmts.html#try

Raising exceptions in your code

Use the **`raise`** statement with an `Exception` instance:

```
if an_error_occurred:
  raise RuntimeError("Spider sense is tingling.")
```

Exercise 1: *(See IPython notebook)*

NumPy and plotting

See IPython notebook:
`02-numpy-and-plotting`

Pandas: tabular data manipulation

See IPython notebook:
`03-pandas.ipynb`

Appendix

Basic types

Basic object types in Python:

- **bool** The class of the two boolean constants `True`, `False`.
- **int** Integer numbers: `1`, `-2`, ...
- **float** Double precision floating-point numbers, e.g.: `3.1415`, `-1e-3`.
- **str** Text (strings of byte-size characters).
- **list** Mutable list of Python objects
- **dict** Key/value mapping

The type of a Python object can be gotten via the `type()` function:

```
In [3]: type('hello')
Out[3]: str
```

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!

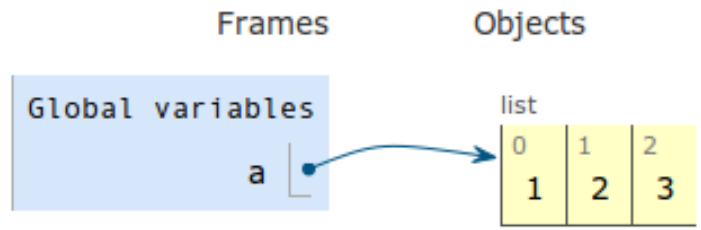
All variables are references (demo)

www.pythontutor.com

All variables are references (demo)

www.pythontutor.com

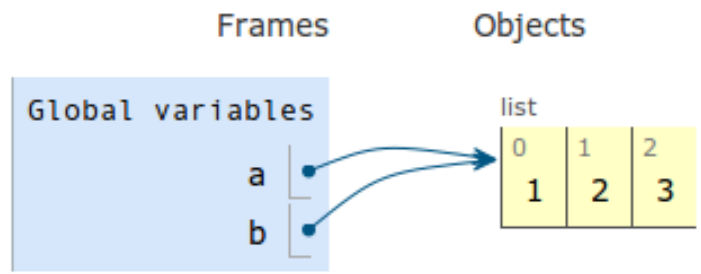
```
1  a = [1, 2, 3]
2  b = a
3  b.remove(2)
4  print a
5  print b
```



All variables are references (demo)

www.pythontutor.com

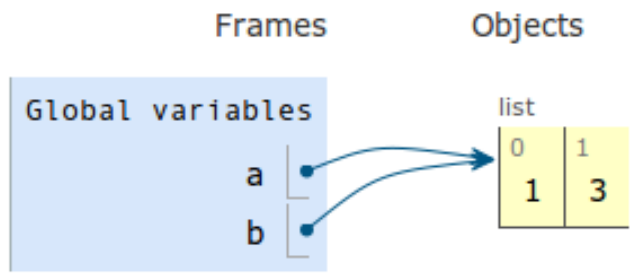
```
1  a = [1, 2, 3]
2  b = a
3  b.remove(2)
4  print a
5  print b
```



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
```



Other containers

The following builtin containers are always available:

dict	*mutable* key/value mapping.
set	*mutable*, unordered set of *unique* elements.
frozenset	*immutable*, unordered set of *unique* elements.

Other specialized containers are available in the `collections` module:

dequeue	a generalization of stacks and queues
namedtuple	similar to a tuple, but allows you to access the elements *by name*
OrderedDict	dictionary that remembers the order that the items were inserted.

Other containers

The following builtin containers are always available:

dict	*mutable* key/value mapping.
set	*mutable*, unordered set of *unique* elements.
frozenset	*immutable*, unordered set of *unique* elements.

Other specialized containers are available in the `collections` module:

dequeue	a generalization of stacks and queues
namedtuple	similar to a tuple, but allows you to access the elements *by name*
OrderedDict	dictionary that remembers the order that the items were inserted.

Sets (1)

The `set` type implements an **unordered** container that holds exactly one object per equivalence class:

```
>>> S = set()
>>> S.add(1)
>>> S.add('two')
>>> S.add(1)
>>> S
set([1, 'two'])
```

Sets (2)

You can create a set and add elements to it in one go:

```
>>> S2 = set([1, 2, 3, 4])
```

and remove elements:

```
>>> S2.remove(2)
>>> S2.pop()
1
>>> S2
set([3,4])
```

Sets (3)

Sets are often used to get unique values from a list:

```
>>> L = [1, 1, 2, 2, 3, 3]
>>> set(L)
set([1, 2, 3])
```

Of course, you can also create a list from a set:

```
>>> S = set((1,2,3))
>>> list(S)
[1, 2, 3]
```

Q: *In what order will the set items appear in the resulting list?*

Sets (3)

Sets are often used to get unique values from a list:

```
>>> L = [1, 1, 2, 2, 3, 3]
>>> set(L)
set([1, 2, 3])
```

Of course, you can also create a list from a set:

```
>>> S = set((1,2,3))
>>> list(S)
[1, 2, 3]
```

Q: *In what order will the set items appear in the resulting list?*

Sets (3)

Sets are often used to get unique values from a list:

```
>>> L = [1, 1, 2, 2, 3, 3]
>>> set(L)
set([1, 2, 3])
```

Of course, you can also create a list from a set:

```
>>> S = set((1,2,3))
>>> list(S)
[1, 2, 3]
```

Q: *In what order will the set items appear in the resulting list?*

Tuples

Tuples are like lists

```
>>> T = (1, 2, 3)
>>> T[0]
1
>>> T[0:1]
(1,)
```

but they are *immutable*

```
>>> T[0] = 'a'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Multiple assignment

You can assing multiple variables at the same time

```
>>> a, b, c = (1, 2, 3)
>>> print(a)
1
>>> print(b)
2
```

It works with any sequence:

```
>>> a, b, c = 'UZH'
>>> print(a)
U
```

Q: *Can you think of a way to swap the values of two variables using this?*

Multiple assignment

You can assing multiple variables at the same time

```
>>> a, b, c = (1, 2, 3)
>>> print(a)
1
>>> print(b)
2
```

It works with any sequence:

```
>>> a, b, c = 'UZH'
>>> print(a)
U
```

Q: *Can you think of a way to swap the values of two variables using this?*

Multiple assignment

You can assing multiple variables at the same time

```
>>> a, b, c = (1, 2, 3)
>>> print(a)
1
>>> print(b)
2
```

It works with any sequence:

```
>>> a, b, c = 'UZH'
>>> print(a)
U
```

Q: *Can you think of a way to swap the values of two variables using this?*

```
>>> a, b = b, a
```

Multiple assignment (2)

Multiple assignment can be used in `for` statements as well.

```
>>> L = [(1,'a'), (2,'b'), (3, 'c')]
>>> for x, y in L:
...     print ("first is " + str(x)
...                + ' and second is ' + y)
```

This is particularly useful with functions that return a tuple. For instance the `enumerate()` function (look it up with `help()`!).

Data structures recap

mutable	**immutable**	
`set`	`frozenset`	unordered container of unique elements
`list`	`tuple`	ordered sequence
`dict`	—	key/values mapping
—	`str`	ordered sequence of characters

How to copy an object?

```
>>> import copy
>>> a = [1, 2]
>>> b = copy.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.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`:

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

What's an *object*?

A Python object is a bundle of variables and functions.

What variable names and functions comprise an object is defined by the object's *class*.

From one class specification, many objects can be *instanciated*. Different instances can assign different values to the object variables.

Variables and functions in an instance are collectively called *instance attributes*; functions are also termed *instance methods*.

Example: the `datetime` object, I

```
>>> from datetime import date
>>> dt1 = date(2012, 9, 28)
>>> dt2 = date(2012, 10, 1)
```

To instanciate an object, call the class name like a function.

Example: the `datetime` object, II

```
>>> dir(dt1)
['__add__', '__class__', ..., 'ctime', 'day',
'fromordinal', 'fromtimestamp', 'isocalendar',
'isoformat', 'isoweekday', 'max', 'min', 'month',
'replace', 'resolution', 'strftime', 'timetuple',
'today', 'toordinal', 'weekday', 'year']
```

The `dir` function can list all objects attributes.

Note there is no distinction between instance variables and methods!

Example: the `datetime` object, III

```
>>> dt1.day
28
>>> dt1.month
9
>>> dt1.year
2012
```

Access to object attributes is done by suffixing the instance name with the attribute name, separated by a dot ".".

Example: the `datetime` object, IV

```
>>> dt1 = date(2012, 9, 28)
>>> dt2 = date(2012, 10, 1)

>>> dt1.day
28
>>> dt2.day
1
```

The same attribute can have different values in different instances!

Instance methods

```
>>> dt1.isoformat()
'2012-09-28'
```

Invoke an instance method just like any other function.

Everything is an object!

The `dir` built-in function is used to list the attributes of an object.

```
>>> dir("hello!")
```

Everything is an object!

The `dir` built-in function is used to list the attributes of an object.

```
>>> dir("hello!")
['__add__', '__class__', '__contains__',
 '__delattr__', '__doc__', '__eq__',
 ...
'strip', 'swapcase', 'title',
'translate', 'upper', 'zfill']
```

. . . a string is an object!

Everything is an object!

```
>>> dir([1,2,3])
['__add__', '__class__', '__contains__',
...
'append', 'count', 'extend',
'index', 'insert', 'pop',
'remove', 'reverse', 'sort']
```

...a `list` is an object!

Everything is an object!

Indeed, you can do:

```
>>> "hello world!".split()
['hello', 'world!']
```

```
>>> [1,1,2,3,5].count(1)
2
```

Everything is an object!

```
>>> dir(1)
```

Everything is an object!

```
>>> dir(1)
['__abs__', '__add__', '__and__',
...
'conjugate', 'denominator',
'imag', 'numerator', 'real']
```

...an `int` is an object!

```
>>> (1).numerator
2
>>> (1).denominator
1
```

Objects *vs* modules

Modules are also namespaces of variables and functions.

The dot operator '.' is also used to access variables and functions from modules. The `dir()` function is also used to list variables and functions from modules.

But each module has *one and only one* instance in a Python program.